UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
ENGENHARIA DE COMPUTAÇÃO

EDUARDO DE MELO LEONARDI

Hardware Implementations of Trellis based Decoders for Linear Block Codes

Final Report presented in partial fulfillment of the requirements for the degree of Computer Engineer

Dipl.-Ing. Stefan Scholl
Advisor

Prof. Dr. Valter Roesler
Coadvisor

Porto Alegre, Dezember 2013

“Success is not the position you stand
but the direction in which you look.”

ACKNOWLEDGEMENTS

I thank my tutor Dipl.-Ing. Stefan Scholl for conducting my work at the University of Kaiserslautern. I would also like to thank Prof.Dr.Ing Nobert Wehn for the amazing work structure of the Microelectronic Systems Design Research Group and my co-advisor Prof.Dr. Valter Roesler for his suggestions which certainly added value to this work.

CONTENTS

LIST OF ABBREVIATIONS AND ACRONYMS

ACS	*Add Compare Select*
APP	*A-Posteriori-Probability*
CS	*Compare Select*
FEC	*Forward Error Correction*
FSM	*Finite State Machine*
FER	*Frame Error Rate*
FPGA	*Field-Programmable Gate Array*
LLR	*Logarithmic Likelihood Ratio*
LUTs	*Look Up Tables*
MAP	*Maximum a Posteriori*
ML	*Maximum Likelihood*
PCM	*Parity Check Matrix*
PN	*Permutation Network*
RAM	*Random Access Memory*
RU	*Recursion Unit*
SISO	*Soft-Input Soft-Output*
SNR	*Signal to Noise Ratio*
TS	*Trellis States*
VA	*Viterbi Algorithm*
VHDL	*VHSIC Hardware Description Language*
VHSIC	*Very High Speed Integrated Circuit*

LIST OF FIGURES

ABSTRACT

Forward error correction based on convolutional codes or block codes is an essential part in today's communication systems. If convolutional codes are used, mostly the graphical trellis representation of a code is used in decoding. Efficient trellis based decoding algorithms can then be used, such as the Viterbi algorithm (VA)[3] or the maximum a posteriori algorithm (MAP)[4].

However, it is shown in [1] that a linear binary block code can also be represented by a Trellis diagram. Then, the efficient VA and MAP can also be applied to block codes.

This work presents two new architectures for the VA and MAP for block codes and their implementation on FPGA. First, we construct a Viterbi decoder and show how a Banyan permutation network can be used to solve the time variance problem of a Trellis diagram for block codes. Afterwards, we use part of the presented to design a MAX-Log-MAP decoder for linear block codes. To our best knowledge they are the first hardware implementations of these kind.

We present implementation details for FPGA designs (Xilinx Virtex 6) of VA and MAP decoders for different trellis sizes. The FPGA designs are analyzed and compared, regarding resource consumption and data throughput. For a 64 state trellis the VA consumes 2800 LUTs and achieves a throughput of 140 Mbit/s, the MAP consumes 6800 LUTs at 70 Mbit/s.

Keywords: FEC, viterbi algorithm, MAP algorithm, block codes.

Implementações em Hardware de Decodificadores baseados em Treliça para Códigos Bloco Lineares

RESUMO

Correção de erro do tipo FEC (do Inglês Forward Error Correction) baseados em códigos bloco ou convolucionais é uma importante parte dos sistemas de comunicação atuais. Se códigos convolucionais são usados, normalmente a representação em treliça do código é utilizada na decodificação. Dessa forma, eficientes algoritmos de decodificação podem ser utilizados, como o algoritmo de Viterbi (VA)[3] e o máximo a posteriori (MAP)[4].

Contudo, é mostrado em [1] que códigos bloco lineares também podem ser representados por um diagrama em treliça. Assim, os algoritmos VA e MAP também podem ser utilizados na sua decodificação.

Esse trabalho apresenta duas novas arquiteturas para o VA e o MAP para códigos bloco e suas implementações em FPGA. Primeiro, nós construímos um decodificador Viterbi e mostramos como uma rede de permutação de Banyan pode ser usada para resolver o problema da variancia no tempo discreto de um diagrama em treliça para códigos bloco. Depois disso, nós reusamos a unidade de recurção do decodificador Viterbi para implementar um decodificador MAX-Log-MAP para códigos bloco. Para o nosso melhor conhecimento, elas são as primeiras implementações em hardware desse tipo.

Nós apresentamos detalhes de implementação em FPGA (Xilinx Virtex 6) do decodificador Viterbi e MAP para diferentes tamanhos de treliça. As implementações em FPGA são analisadas e comparadas, considerando o uso de recursos e vazão de dados. Para um diagrama em treliça com 64 estados, o VA consome 2800 LUTs com uma vazão de 140 Mbit/s. Já o MAP consome 6800 Luts a 70 Mbit/s.

Palavras-chave: FEC, Viterbi, MAP.

1 INTRODUCTION

In recent years, there has been an increasing demand to reliably transmit data over noisy communication channels at high transmission rates. Shannon stated that by using error correcting codes, it is possible to reliably transmit data over noisy channels, as long as the information rate is lower than the channel capacity. The error codes add redundancy to the input message and exploit this redundancy when decoding the received message. The aim of channel coding is to find error correcting codes that allow quick and reliable transmission of data.

1.1 Motivation

Two important error correcting codes exist to transmit data over noisy channels: block codes and convolutional codes. An important difference between these two codes is that if convolutional codes are used, usually the graphical trellis representation is used in decoding. Efficient trellis based decoding algorithms can than be used, such as the Viterbi Algorithm (VA)[3] and the Maximum a Posteriori (MAP)[4] algorithm. These both algorithms applied to convolutional codes are particulary suitable for implementations in hardware. The VA performs Maximum Likelihood (ML) decoding and outputs the most probably sent codeword. According to the common literature, ML decoding achieves the best possible error rates.

ML decoding for block codes is achievable by using word correlating decoders. But this method is inefficient and even intractable for large codes. Back in 1978, Jack Wolf wrote a paper [1] showing that soft decision ML decoding of any (n,k) linear binary block code can be accomplished by using the Viterbi algorithm [3] applied to a Trellis diagram with no more than 2^{n-k} states, called Wolf's trellis diagram. Thus, the ML decoding complexity for block codes can be reduced. To our best knowledge, this idea has never been explored to construct a physical hardware.

The aim of this thesis is to present hardware architectures of Trellis based decoders for block codes. Trellis decoders for block codes have many different applications. Some of their use cases are:

1. as a maximum likelihood decoder for small block codes.

2. as a component decoder for turbo product codes [5].

3. as a check node decoder for generalized LDPC codes [6].

4. as a component for soft decision decoding of Reed-Solomon codes [7] [8].

The Wolf's trellis diagram is first used to develop an architecture for a Viterbi decoder.

In more powerful decoding systems, the concept of feedback - a well-known technique in electronics – is implemented between the two component decoders The use of feedback requires the existence of Soft-Input Soft-Output (SISO) decoding algorithms for both component codes. An application example for SISO decoding are block turbo codes [5] [9]. The Viterbi algorithm outputs the most likely codeword sent, but does not output any information on the reliability of the decisions made. Thus, the VA is not suitable for turbo code applications. The second decoder presented in this thesis is a MAX-Log-MAP decoder for block codes, a soft output decoder which makes estimation of bits based on the whole received block [4].

Structure of the thesis.

We first give a general overview on the basics of channel coding in Section 2. Section 3 contains decoding algorithms for linear block codes. The decoders' architectures and implementation issues are given in Section 4. Section 5 contains the implementation results. Finally, Section 6 presents our conclusions and an outlook on future work.

2 CHANNEL CODING

This thesis deals with the topic of hardware implementations of trellis based decoders for linear block codes. Therefore, basic knowledge of channel coding theory is needed for its understanding. We introduce the basic concepts of channel coding in this Section. The first section presents a brief introduction to communication systems. In the following sections, we will discuss the structure of block codes and convolutional codes, as well as their differences.

2.1 Basics

Figure 2.1 shows a generic simplified communication system. The information source produces the message m to be transmitted. One important function of channel coding is Forward Error Correction (FEC). In order to combat the noisy environments through which the data must be transmitted, the channel encoder introduces, in a controlled manner, redundant bits to the message m. The output of the encoder is an encoded sequence c called codeword. Other important functions of channel coding are channel measurement and a more uniform distribution of errors through the use of interleavers. However, in this work we are concerned only with the FEC function of channel coding.

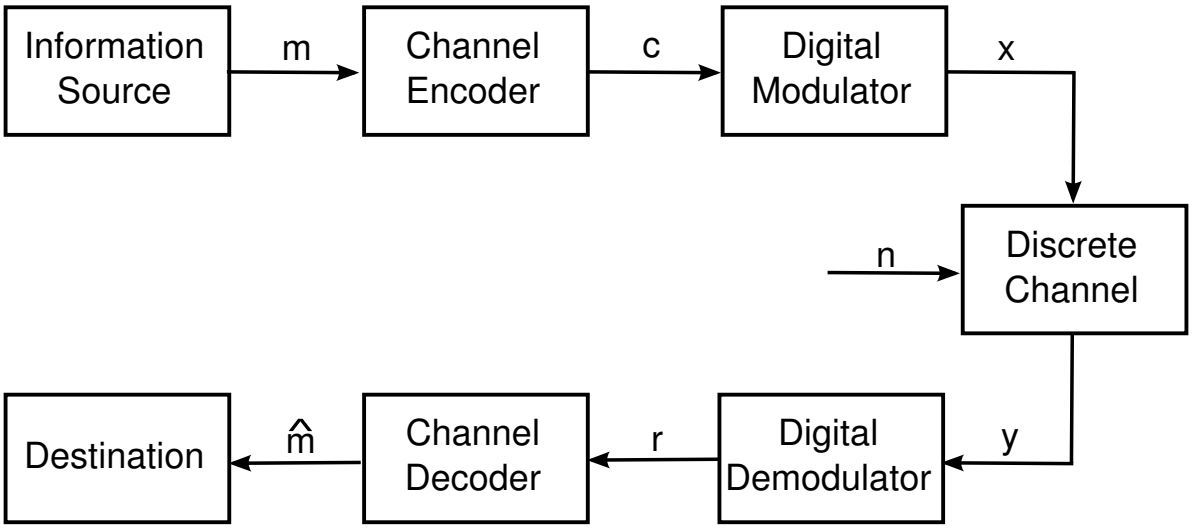

Figure 2.1: A digital communication system.

The discrete symbols of the codewords are not suitable to be transmitted over a physical channel. Thus, the modulator maps these discrete symbols to modulated symbols t

called the transmit sequence, which is suitable for the channel. Many kinds of modulation schemes exist. One of the most common ones, the BPSK, is assumed along this thesis.

The modulated symbols enter the channel and are affected by random noise n. Different types of noise disturbances exist and each channel is subject to many of them. Defining n as an additive white noise Gaussian variable with zero mean and variance σ^2, the output of the channel can be expressed as:

$$y = x + n \tag{2.1}$$

The demodulator processes the received symbols y and transforms them into a received sequence r.

For each code type different decoding techniques exist. The channel decoder analyzes the received sequence and tries to overcome the signal degradation introduced by the channel. For this, the decoder uses the redundant information introduced by the encoder. The choice of the decoding strategy is dependent on the application at issue and the noise characteristics of the channel.

The decoder delivers the estimated message $\hat{m}$ correspondent to the chosen codeword c to its final destination. Ideally, $\hat{m}$ should be equal to m, but the noise introduced by the channel might cause some decoding errors.

2.2 Block Codes

In block codes, the data is encoded into blocks. An (n, κ) block code over the finite field $GF(q)$ is a set of q^κ n-tuples called codewords. An important property of a linear block code it forms a k-dimensional vector space. This property allows a more compact representation for the code, as we will show in Section 2.2.1.

Definitions:

1. Let Σ_k be a vector over the finite field $GF(q)$, containing the k-tuple messages m of a block code and Σ_n be an alphabet over the same field containing blocks of length n.

2. The Hamming distance between two strings of equal length is the number of positions at which the corresponding symbols are different.

3. The Hamming weight of a codeword is equal to the non-zero components of the codeword.

4. The minimum Hamming weight ω_{min} of a code is the smallest Hamming weight of any non-zero codeword $c \in C$.

Associated with the code is an encoder. The encoding function is an injective mapping $C : \Sigma_\kappa \rightarrow \Sigma_n$ which encodes each message m_i individually to a different codeword c_i. There should be an one-to-one correspondence between a message m and its codeword. Hence, among all the 2^n words $w \in \Sigma_n$, only 2^k are codewords. The length of the code is the number n, whereas dimension of the code is called κ. The code rate is $R = \kappa/n$ and designates the percentage of information bits transmitted in relation to the total number of bits.

An important property of a block code is its minimum distance d_{min}. It is defined as the minimum number of amendments which may transform one codeword into another. More formally, the minimum distance can be expressed as:

$$d_{min} = \min_{m_1, m_2 \in \Sigma_k, m_1 \neq m_2} \Delta(C(m_1), C(m_2)) \tag{2.2}$$

where C is the encoding function and $\Delta(C(m_1), C(m_2))$ denotes the Hamming distance between the codewords c_1 and c_2.

The minimum distance is a measurement of how capable the code is of detecting or correcting errors. A code with minimum distance d_{min} is capable of detecting $(d_{min} - 1)$ errors or correcting $(d_{min} - 1)/2$ errors. An easier way to find the code's minimum distance is to take its minimum Hamming weight, since a linear block code satisfies $d_{min} = w_{min}$.

2.2.1 Generator Matrix

A block code can be represented as a list. For large κ, this representation is too complex to store and decode. Here, we present a more compact representation for a linear block code.

Since a block code is a κ-dimensional vector space, a set of κ linearly independent vectors $g_0, g_1, ..., g_{\kappa-1}$ of length n exists, so that every codeword $c \in C$ is a linear combination of these vectors [10]:

$$c = m_0 g_0 + m_1 g_1 + ... + m_{\kappa-1} g_{\kappa-1}, \tag{2.3}$$

where $m_i \in GF(q)$ are constants and all the arithmetic is done over modulo q. Thinking of g_i as a rows of a matrix:

$$G = \begin{bmatrix} g_0 \\ g_1 \\ \vdots \\ g_{\kappa-1} \end{bmatrix} \tag{2.4}$$

and letting $m = [m_1, m_2, ..., m_{\kappa-1}]$ be a message, from equation 2.3 we define an encoding operation for block codes as:

$$c = mG \tag{2.5}$$

Every codeword $c \in C$ can be represented as a multiplication of a vector m with the matrix G. Since the rows of G generates the (n, κ) code C, G is called the *generator matrix* of C. Representing a code thus requires storing only κ vectors of length n, instead of storing all the 2^κ codewords.

2.2.2 Parity Check Matrix

The Parity Check Matrix (PCM) is especially important for the decoding process of linear block codes. It contains information on the redundant bits and is used in many decoding algorithms for block codes. We now present how to derivate such a matrix.

A Parity Check Matrix H for a code C is obtained by taking the generator matrix of its dual code $C^\perp$ [10].

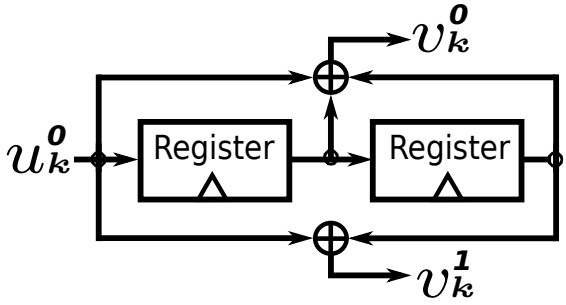

Figure 2.2: A convolutional encoder.

As $C^{\perp}$ is a vector space with dimension $n-\kappa$ and a basis denoted by $h_0, h_1, ..., h_{n-\kappa-1}$, we build the matrix H by using these basis vectors as rows:

$$H = \begin{bmatrix} h_0 \\ h_1 \\ \vdots \\ h_{n-\kappa-1} \end{bmatrix} \tag{2.6}$$

The *generator matrix* and the *parity check matrix* for a linear code satisfy:

$$GH^T = 0 \tag{2.7}$$

Hence, the parity check matrix for a code can be determined from its generator matrix and vice versa. Moreover, a vector $v \in \Sigma_n$ is a codeword of C, if and only if:

$$vH^T = 0 \tag{2.8}$$

We use Equation 2.8 to derive a trellis diagram for block codes in section 3.3.

2.3 Convolutional Codes

Although the decoding of convolutional codes is not the topic of this thesis, it is convenient to present the differences between block codes and convolutional codes. Convolutional codes are widely used and their decoding implementation issues are already known. In addition, part of the work presented here is based on trellis based decoders for convolutional codes, especially the work presented in [2].

2.3.1 Introduction to Convolutional Codes

In the following, we will give a brief introduction to convolutional codes.

In a convolutional code, at each time step, a stream information sequence u is divided into groups of κ information bits $u_k = \{u_k^0, u_k^1, ..., u_k^{\kappa-1}\}$ which are encoded to code bits $v_k = \{v_k^0, v_k^1, ..., v_k^{n-1}\}$ of length n, with k being a step time.

A convolutional encoder has m memory elements (registers) which stores data from the past bits. Thus, the output of the decoder depends on $m+1$ past bits. The encoding is made by the convolution of the input stream with the encoder's impulse responses. Each one of the n impulse responses is associated to a generating polynomial $\{g_0, ..., g_n\}$ of maximal degree m.

A Mealy Finite State Machine (FSM) is the most common representation of a convolutional encoder. For the encoder of Figure 2.2, the FSM is shown in Figure 2.3.

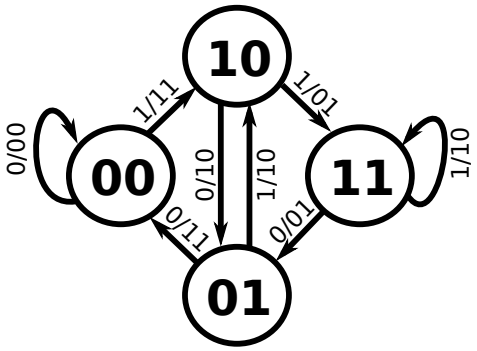

Figure 2.3: The FSM representation of a convolutional encoder.

Convolutional Codes are not the main scope of this thesis. For those who have interest in learning more about their operation, we provide an example of encoding and decoding using convolution codes in Appendix A.3.

2.3.2 The Difference between Block Codes and Convolutional Codes

Convolutional decoders work with streams of data which conceptually can be infinitely long. In practice, the stream is truncated and transmitted in blocks of fixed length. Block code decoders work with the so-called block of codewords. Each block of data to be transmitted has a specific length n.

In comparison to the convolutional codes, where the code performance is function of the number of its memory elements m and the error correction is possible given the illegal state transitions, for block codes, the code performance is function of its minimum distance d_{min} and the error correction is made based on the fact that not every received string $v \in \Sigma_n$ is a valid codeword.

Trellis Diagram:

We will now discuss the differences between Trellis diagrams for block codes and for convolutional codes. The Trellis diagram is an important tool for channel decoding. We will give further information concerning the Trellis diagram in Section 3.3.

We obtain a Trellis diagram for convolutional codes by unrolling the encoder's state machine over discrete time. Such a diagram presents all the possible state transitions of the convolutional encoder. Figure 2.4 shows a Trellis diagram for the state machine of Figure 2.3. In Section 3.3, we present a method for constructing a Trellis diagram for block codes. Such a diagram is a compact method of representing all of the code's codewords, in which every distinct path through the Trellis represents a different codeword.

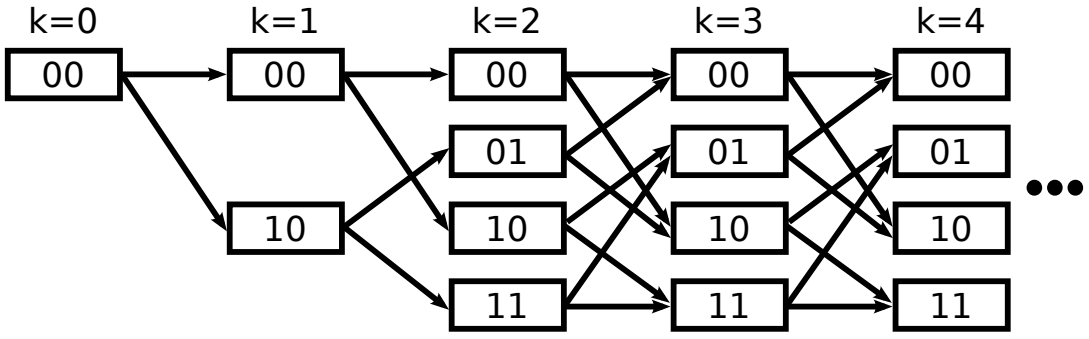

Figure 2.4: A Trellis diagram for convolutional codes.

The structure of a Trellis diagram for convolutional codes is very regular. By examining the structure of the Trellis diagram shown in Figure 2.4 step-by-step, the same transitions can be observed. For block codes, the structure of the trellis is time variant [1]. The transitions change at each time step. Therefore, the decoding of block codes using a Trellis diagram is more complex. In Section 4 we present architectures of block code decoders and explain how to overcome this problem.

3 DECODING OF LINEAR BLOCK CODES

This Section presents three different decoding algorithms for linear block codes. We start by discussing the importance of soft-input decoding.

3.1 Soft-input Decoding

As shown in Section 2.1, the transmit sequence t is disturbed by a white noise Gaussian variable n with zero mean and variance σ^2.

Following [2], for an AWGN channel and considering a BPSK modulator, which maps the binary symbols $c_k \in \{0, 1\}$ to modulated symbols $t_k \in \{-1, 1\}$, with $t_k = 1 - 2c_k$, the probability density function (PDF) for each output symbol results in:

$$p(y_k|t_k) = \frac{1}{\sqrt{2\pi\sigma^2}} exp(-\frac{(y_k - t_k)^2}{2\sigma^2}) \tag{3.1}$$

Instead of using hard decisions, where each received bit is considered to be definitely one or zero, we make the decoding based on soft decision. Soft decision decoding means that the decoding process uses channel measurement information, i.e., it makes use of the probability that every received code symbol has to be either one or zero. This probability comes from the PDF and is expressed as a *logarithmic likelihood ratio* (LLR) [10]:

$$\lambda_k(y_k|t_k) = \ln \frac{Pr(y_k|t_k = 1)}{Pr(y_k|t_k = -1)} = \ln \frac{Pr(y_k|t_k = 1)}{1 - Pr(y_k|t_k = 1)} \tag{3.2}$$

Alternatively, a LLR can be expressed in its inverse form:

$$\tilde{\lambda_k}(y_k|t_k) = \ln \frac{Pr(y_k|t_k = -1)}{Pr(y_k|t_k = 1)} \tag{3.3}$$

The way that the LLRs are defined affects the way that the decisions of the decoding algorithms are made. We use the LLR definition as in Equation 3.2 through this thesis.

Figure 3.1 shows λ_k as function of $Pr(y_k|t_k = 1)$. The sign of λ_k is the hard decision of y_k while $|\lambda_k|$ is a measure of reliability.

For better understanding of the LLRs consider the two PDFs plotted in Figure 3.2. It shows one PDF for each possible modulated symbol t_k considering a BPSK modulator. If the signal degradation is high enough, it can happen that one modulated symbol moves toward the other one and slightly crosses the y-axis ($x = 0$). In this scenario, the hard decision of this received symbol would be the other symbol (not the one sent). But as $Pr(y_k|t_k)$ would be close to $1/2$, the corresponding LLR and thus the reliability of the

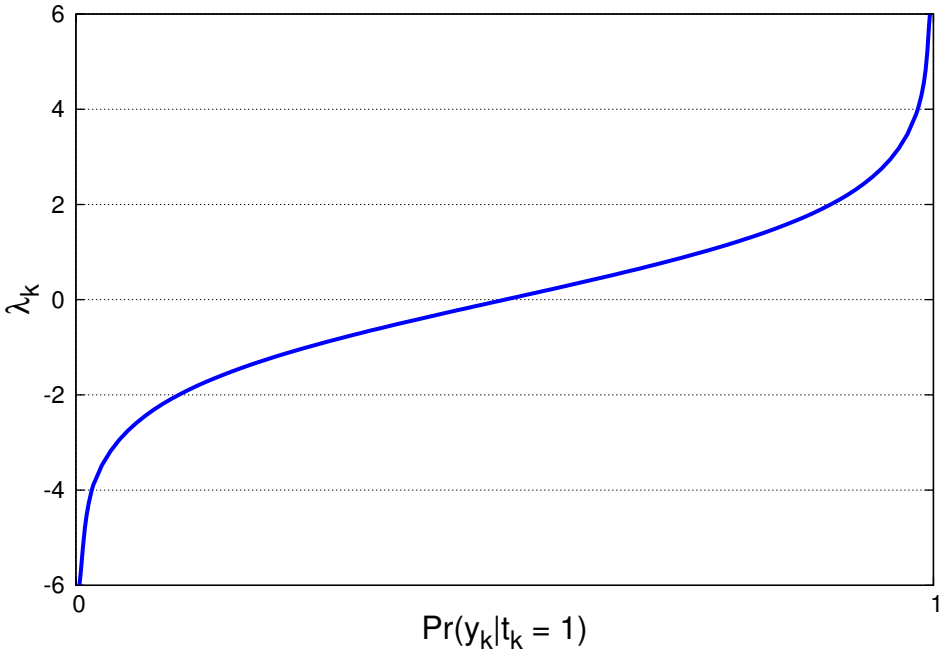

Figure 3.1: LLR as function of $Pr(y_k|t_k = 1)$.

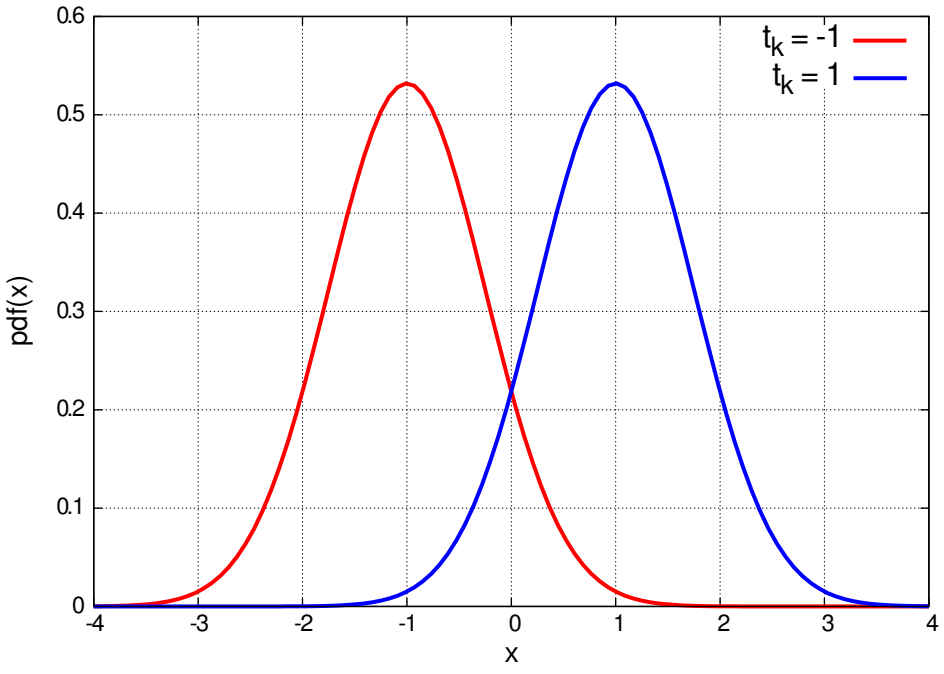

Figure 3.2: PDFs for each modulated symbol using a BPSK

received symbol would be close to zero. The higher the variance of the random noise variable is, the more spread these curves are over the x-axis.

In case of soft-input decisions, the received symbols r_k assume arbitrary values. Otherwise they assume values in the set $\{-1, 1\}$. A lot of information that can be used in favor of the decoder is lost if hard-decisions are used. The use of LLRs gives the decoder a measure of how reliable each bit is.

3.2 The Word Correlating Decoder

Given a list with all the 2^k codewords of a code, the correlation decoder solves the ML criteria by finding the codeword c that maximizes the correlation between a codeword c and the received sequence r:

$$\max_{c \in C} Pr(y|c) = \max_{c \in C} \prod_{k=0}^{n-1} Pr(y_k|c_k) \tag{3.4}$$

Alternatively, it is also possible to maximize its logarithm:

$$\max_{c \in C} \ln Pr(y|c) = \max_{c \in C} \sum_{k=0}^{n-1} \ln Pr(y_k|c_k), \tag{3.5}$$

Using the definition of LLR given in section 3.1, with r_k being an LLR, the chosen codeword is the one that minimizes:

$$\sum_{k=0}^{n-1} r_k c_k \tag{3.6}$$

The decoder must then compare every codeword with the input LLRs. Hence, its computation complexity is intractable for large k. The Viterbi Algorithm (VA) organizes the computation in a more efficient recursive form. We used the word correlating decoder in this thesis only to test the results generated by the VA, as they both solve the same problem.

3.3 The Viterbi Algorithm

Before we proceed with the explanation of the Viterbi algorithm, we will describe how to construct a Trellis diagram for linear block codes. The Viterbi algorithm uses such a diagram to find the most likely sent codeword c given the input LLRs.

3.3.1 Trellis Diagram for Linear Block Codes

There is a graph associated with a block code. This graph is called Wolf Trellis for the code. All the paths through the Trellis correspond to the words v that satisfy the parity check condition: $vH^T = 0$. In this section, we will describe a practical way of how to build a trellis diagram for binary block codes with parity check matrix H. A more general and detailed construction of a trellis diagram for block codes over $GF(q)$ is presented in [1].

A trellis for block codes is a collection of nodes belonging to states $S = \{S_0, S_1, ..., S_{2^{n-\kappa}-1}\}$ grouped into sets indexed by k. $S_{i,k}$ denote a node from a state i at step k.

Let H_k denote the $k - th$ column of H, with H_1 being the first column.

The construction algorithm then is as follows:

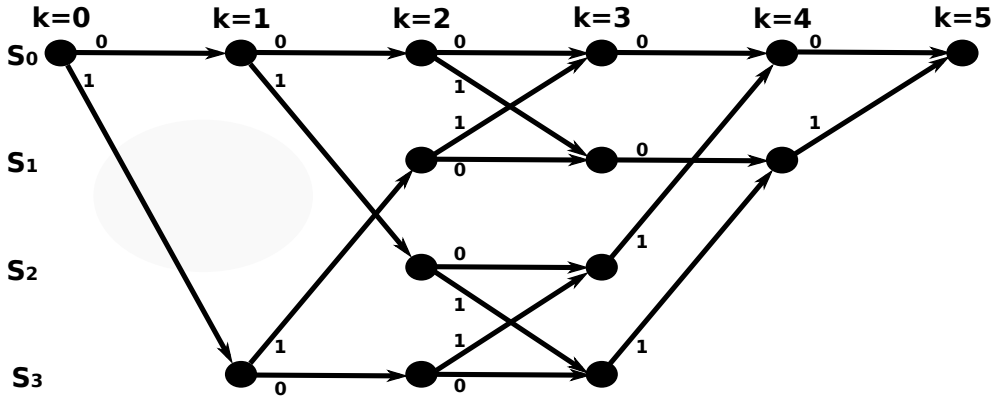

Figure 3.3: A Trellis diagram for a linear block code [1].

1. At depth $k = 0$ there is only one node $S_{0,0}$.

2. For each step $k = \{0, 1, ..., n\}$, the collection of nodes in depth $(k + 1)$ as well as the connections among the states are calculated from the nodes at depth k for each unidirectional connection $\alpha_j \in \{0, 1\}$, by using the following formula:

$$S_{l,k+1} = S_{i,k} \oplus \alpha_j H_{k+1} \tag{3.7}$$

   The above formula shows that considering binary block codes, for zero-transition the state is maintained, while for one-transition the next state is calculated based on the columns of H.

3. We remove the nodes that do not have a path to the all-zero state at depth n, $S_{0,n}$, as well as the lines drawn to this nodes.

For a code with *parity check matrix*:

$$H = \begin{bmatrix} 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \end{bmatrix} = \begin{bmatrix} h_1 & h_2 & h_3 & h_4 & h_5 \end{bmatrix} \tag{3.8}$$

the corresponding Wolf Trellis diagram is shown in Figure 3.3.

3.3.2 The Viterbi Algorithm Using a Trellis Diagram.

A codeword c corresponds to a path through the Trellis. Due to the noise introduced by the channel, the received sequence r may not correspond to a codeword. The VA finds the path through the Trellis which is the closest to the received sequence r.

The Viterbi algorithm solves the maximum likelihood criteria presented in Section 3.2:

$$\max_{c \in C} \sum_{k=0}^{n-1} \ln Pr(y_k|c_k) = \min_{c \in C} \sum_{k=0}^{n-1} \gamma(y_k|c_k), \tag{3.9}$$

where $\gamma(y_k|c_k)$ is a *branch metric*.

The branch metrics are the weights of the Trellis diagram's transitions and they are used in the Viterbi algorithm's decisions. A branch metric $\gamma_{k,k+1}^{i,l}$ is assigned to each possible state transition at time step k: $S_{i,k} \rightarrow S_{l,k+1}$.

If on the one hand, a trellis diagram for block codes is more complex, the calculation of its branch metrics on the other hand is much simpler.

Branch Metrics

Given a received channel symbol r_k, obtained from a transmitted bit c_k which is the corresponding output of a state transition represented by a trellis diagram, considering an AWGN channel, for linear binary block codes, the branch metrics calculation is:

$$\gamma_{k,k+1}^{i,l} = c_k r_k = \begin{cases} r_k, & \text{if } i \neq l \\ 0, & \text{otherwise} \end{cases} \tag{3.10}$$

The above formula shows that the branch metrics of all transitions which mantain the state (zero-transitions) are zero, while the branch metrics of the one-transitions are the current input symbol (LLR) at step k.

Viterbi Algorithm

The Viterbi algorithm [3] is comprised of two parts: a forward recursion and a traceback. The forward recursion accumulates probabilities for all states based on the current input symbol by using the state transitions represented by a trellis diagram. The traceback part reconstructs the original data, once a path through the trellis is identified.

Definitions:

1. A path to a state i at step k is the collection of all the k decisions made up to the time k that lead to the state i.
2. A state metric at a step k is a measure of how good the path that leads to this state is in comparison to the paths that lead to the other states. Hence, only the difference between the state metrics and the current input symbol that influence the decisions of the VA.
3. Let $\alpha_{i,k}$ denote the state metric of a node $S_{i,k}$.
4. The metric of the first node in Trellis $S_{0,0}$ is zero: $\alpha_{0,0} = 0$.

During the forward recursion, at each decoding cycle the paths with the least sum of branch metrics, called the *local survivors*, are selected by using the following formula:

$$\alpha_{l,k+1} = \min(\alpha_{l,k}, \alpha_{i,k} + \gamma_{k,k+1}^{i,l}) \tag{3.11}$$

The state metrics S_{k+1} are updated based on the previous state metrics S_k and the current input symbol r_k.

If the reader is not familiar with the operation of the VA, we suggest having a look at the example provided in Appendix A.1.

The decision bits $dec_{i,k+1}$ generated for each state in Equation 3.11 are stored in a *survivor memory*. At the end of the forward recursion, the most likely sequence through the Trellis is identified.

The traceback algorithm reads the local survivors from the survivor memory in order to extract the most likely sequence. Starting with the final state in Trellis $S_{0,n}$, the decision bit generated for this state, $dec_{0,n}$, is retrieved from the survivor memory and the preceding state $S_{i,n-1}$ is derived based on the bit read. The decision bit associated with this new state is also read and so forth. The backward operation read sequence of decision bits is the most likely codeword sent, given the received sequence r.

3.4 The (MAX)-Log-MAP Algorithm

In concatenated coding system, the overall performance of the system is increased if both decoders use soft-input values. The Viterbi algorithm is a hard-output ML sequence detection algorithm that does not output any information on the reliability of the decisions made. The logarithmic *maximum-a-posteriori* (Log-MAP) is a soft-input, soft-output algorithm that makes estimation of bits based on the whole received sequence r.

The Log-MAP is based on an algorithm proposed by Bahl, Cocke, Jelinek and Raviv, the BCJR algorithm [4]. It computes the A-Posteriori-Probability (APP) Logarithmic Likelihood Ratio (LLR) for each sent bit c_k as:

$$\Lambda(c_k) = \ln \frac{Pr(c_k = 1|r)}{Pr(c_k = 0|r)} \tag{3.12}$$

As described in [11], the calculation of 3.12 in the probability domain uses a lot of multiplications and additions. For hardware implementations it is preferable to port the calculations to the logarithmic domain.

Exploiting the idea of the Jacobian logarithm:

$$\begin{aligned} \ln(e^{\delta_1} - e^{\delta_2}) &= min^*(\delta_1, \delta_2) \\ min^*(\delta_1, \delta_2) &= min(\delta_1, \delta_2) - \ln(1 + e^{-|\delta_2 - \delta_1|}), \end{aligned} \tag{3.13}$$

where $\ln(1 + e^{-|\delta_2 - \delta_1|})$ is a correction term often referred to as $f_c(|\delta_2 - \delta_1|)$. The basic idea of the Log-MAP Algorithm is to transform the multiplications into additions and the additions into minimum selections with additional correction terms.

Using a Trellis diagram as a basis, the APP LLR in the logarithmic domain can be calculated by using three metrics:

$$\begin{aligned} \ln \frac{Pr(c_k = 1|r)}{Pr(c_k = 0|r)} &= \min_{\forall(i,l)} *(\gamma^{i,l}_{k,k+1}(c_k = 1) + \alpha_{i,k} + \beta_{l,k+1}) \\ &\quad - \min_{\forall(i,l)} *(\gamma^{i,l}_{k,k+1}(c_k = 0) + \alpha_{i,k} + \beta_{l,k+1}), \end{aligned} \tag{3.14}$$

where i is the index of the current state in trellis and l is the index of the next state connected by a one or zero-transition. The metrics $\alpha_{i,k}$ and $\beta_{l,k+1}$ refer to state metrics and $\gamma^{i,l}_{k,k+1}$ are branch metrics. Note that $\gamma^{i,l}_{k,k+1}(c_k = 1)$ refers to metrics of one-transitions and that $\gamma^{i,l}_{k,k+1}(c_k = 0) = 0$ for block codes.

The α and β metrics are computed in a forward and backward recursion, respectively. The α metrics are the same computed in Equation 3.11 during the VA's forward recursion. The β metrics are computed in a similar way, but beginning with the last state in the Trellis $S_{0,n}$:

$$\beta_{i,k} = \min(\beta_{i,k+1}, \beta_{l,k+1} + \gamma^{i,l}_{k,k+1}) \tag{3.15}$$

Knowledge of the whole input sequence is needed for the calculation of each individual bit. The α-metrics contain information on all the branch metrics from the start of the Trellis up to time step k. Figure 3.4 shows the three metrics used in the calculation of Λ_k. The branch metrics from the successor state $(k + 1)$ until the end of the Trellis are contained in the β-metrics. The only metrics that are not used, neither for the α- nor for the β-calculation, are $\gamma^{i,l}_{k,k+1}$. These metrics are directly used in the LLR calculation 3.14.

The arithmetic complexity can be further reduced by omitting the correction terms $f_c(|\delta_2 - \delta_1|)$ in Equation 3.13. The resulting algorithm is then called the MAX-Log-MAP

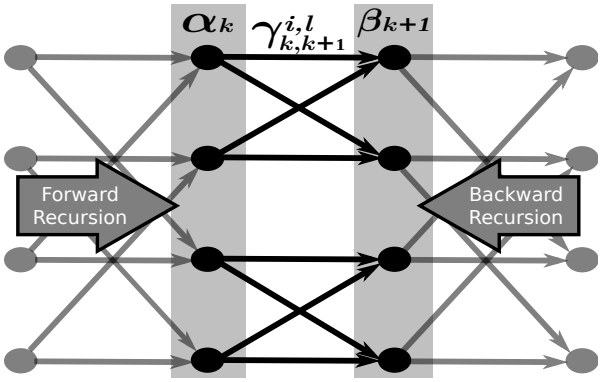

Figure 3.4: Metrics used in the LLR calculation of Λ_k [2].

algorithm. The omission of the correcting terms leads to a slight loss in communication performance in turbo code applications [2]. Since the complexity of the MAX-Log-MAP is smaller than the Log-MAP's, the MAX-Log-MAP is the most used implementation.

The operation of the Max-log-MAP algorithm is difficult to understand by only looking to the formulas above. Therefore we provide a Max-log-MAP algorithm example in Appendix A.2.

4 BLOCK CODES DECODER ARCHITECTURES

We presented decoding algorithms for linear block codes in Section 3. In this section, we give an overview of the general architectures of the Viterbi and the MAX-Log-MAP decoder for linear block codes. The first section introduces the architectures of each building block and the implementation issues of the Viterbi decoder. After that, we use part of the information presented to derive the soft-output of the MAX-Log-MAP decoder.

4.1 The Viterbi Decoder

We explained the Viterbi algorithm in section 3.3.2. In this section, a general overview on the decoder basic building blocks is given.

Figure 4.1 shows a block diagram of a Viterbi decoder. The LLR memory stores the LLRs of each received bit r_k. They are fed into the Recursion Unit (RU), which processes the branch and state metrics during the *add compare select* (ACS) recursion. The state metrics that need to be compared during the ACS recursion change dynamically. The comparisons are specified by the columns h_k of the Parity Check Matrix (PCM). Therefore, the PCM memory stores the whole Parity Check Matrix of the code being decoded. The decision bits $dec_{i,k}$ generated by the RU for each state $S_{i,k}$ are stored in the survival memory. In the Survival Management Unit (SMU), a traceback algorithm retrieves the stored information in order to decode the most likely path through the Trellis. During the traceback, the previous states in Trellis have to be derived and again information on the code's PCM is used.

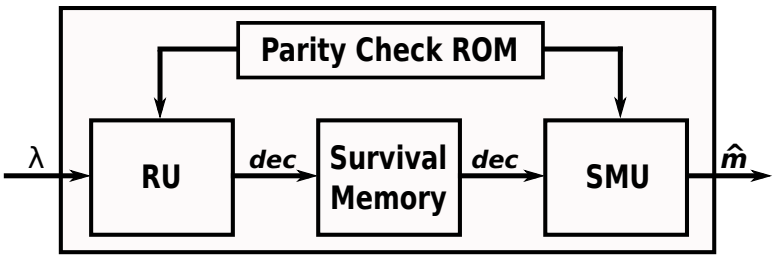

Figure 4.1: Block diagram of a Trellis decoder for block codes.

4.2 Recursion Unit

A Trellis diagram for block codes, or Wolf Trellis diagram, has the property that one of the two branches that leave a node $S_{i,k}$ always leads to a node $S_{i,k+1}$ from the same state i. The other branch that leads to the node $S_{i,k+1}$ comes from another node $S_{l,k}$.

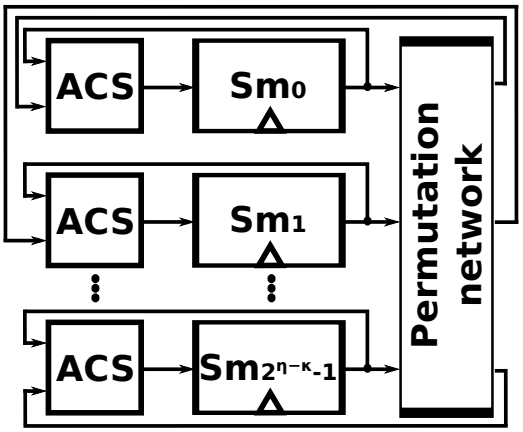

Figure 4.2: Recursion unit for a generic block code.

The number l, which is the metric to be compared, changes after each time step and it is function of the actual column of H. This means that during the ACS recursion, one of the two state metrics to be compared is the same metric from the previous time step, while a permutation network selects the other metric.

Figure 4.2 shows the recursion unit's architecture. All the state metrics of one Trellis step in this architecture are processed in parallel. The recursion unit calculates the new state metrics based on the previous state metrics and the current input symbols. It is composed of $2^{n-\kappa}$ ACS units, a permutation network and $2^{n-\kappa}$ state metrics registers, which accumulate the state metrics cycle by cycle. It also outputs the $2^{n-\kappa}$ decision bits of each state every decoding cycle.

It is important to say that this architecture also calculates state metrics and decision bits for the nodes that do not exist in the original trellis diagram construction of Section 3.3.1. Nevertheless, the decoder stills work properly if we consider two things:

1. If the state metrics from the first step in the Trellis are initialized as:

$$\begin{aligned} S_{0,0} &= 0, \\ S_{i,0} &= \infty, 1 \leq i \leq 2^{n-\kappa} - 1 \end{aligned} \tag{4.1}$$

   only the paths starting from the state $S_{0,0}$ are considered. In practice, ∞ is a value that gives a sufficient low probability for these states. In Section 4.6 we show how to derive this value.

2. Decoding is also possible without expurgating the nodes that do not have a path to the last state in trellis, $S_{0,n}$. Even if the decision bits for these nodes are calculated, the traceback algorithm, which will be presented in Section 4.3, does not consider them.

ACS Unit

The ACS modules compute the minimum selection of Equation 3.11:

$$\alpha_{l,k+1} = \min(\alpha_{l,k}, \alpha_{i,k} + \gamma_{k,k+1}^{i,l}).$$

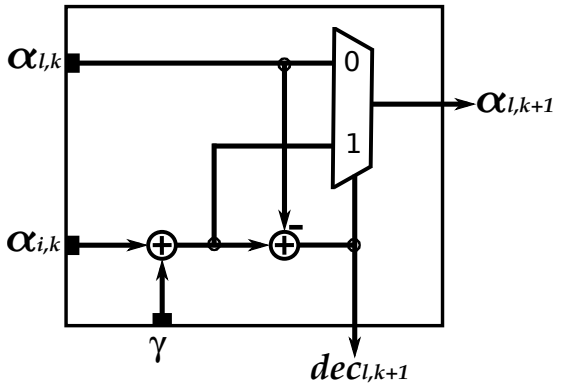

Figure 4.3: ACS unit

Figure 4.3 shows the architecture of the ACS unit. The branch metric γ is added to the metric to be compared $\alpha_{i,k}$. A subtraction compares this result with the other state metric $\alpha_{l,k}$. Note that nothing needs to be added to the other state metric, since the branch metrics of all zero-transitions are zero. Finally, the sign of the subtraction, which is also the decision bit of the next state $S_{l,k+1}$, is used to select the least state metric. All the decision bits are stored in the ($2^{n-\kappa}$ x n) survival memory.

Permutation Network

Since the state metrics to be compared changes every decoding cycle, we need to arrange the state metrics data before calculating the ACS operation. A Permutation Network (PN) is a switch based network capable of realizing permutations of its inputs to its outputs. The building blocks of these networks are switches capable of permuting their two input terminals to their two output terminal. Each switch can be implemented with two multiplexers. A control signal is used to either permute or not permute the two inputs of the switch (see Figure 4.4).

The Benes permutation network.

The Benes permutation network [12] is capable of realizing all the possible $n!$ permutations of its n inputs to its n outputs. Because the network is constructed in a recursive form, the number of inputs is a power of two.

The total number of switches used to implement a Benes PN with n inputs is $n\ln(n) - n/2$. By Equation 3.7, the columns of the parity check matrix decide which nodes of the Trellis are connected through a one-transition. The columns must then control the behavior of the network and select the metric to be compared. Hence, the control of each switch in the network is a logic function of these columns.

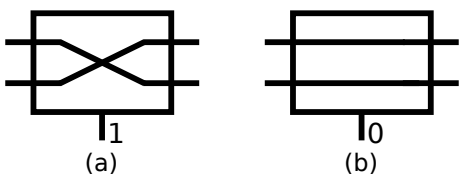

Figure 4.4: Control of the switches. (a) Control = 1 $\rightarrow$ swap inputs. (b)Control = 0 $\rightarrow$ do not swap inputs.

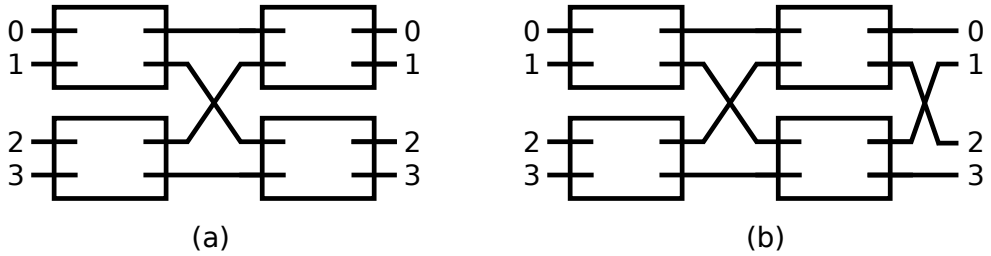

Figure 4.5: Two permutation networks with 4 inputs. (a) Butterfly permutation network. (b) Banyan permutation network.

Considering that the maximum number of permutation necessaries in this application is $2^{n-\kappa}$ out of the $(2^{n-\kappa})!$ that the Benes PN performs, using a Benes PN would be a waste of resources. Besides that, its elevated number of stages adds more latency to the ACS recursion's critical path. In addition, a complex control logic has to be used to control each switch of the network. We shall use a smaller PN with a simpler control.

The Banyan permutation network.

We construct the Banyan permutation network [13] from a butterfly network by amending its final part (See Figure 4.5).

The smallest butterfly network is composed of a single switch. We construct the butterfly network in a recursive form. A network with n inputs is obtained from 2 butterfly networks with $n/2$ inputs, i.e., two butterfly sub-networks.

We place the second sub-network below the first one and numerate the outputs of the first sub-networks from 0 to $n/2 - 1$ and of the second one from $n/2$ to $n - 1$. S(i) denotes the $i - th$ output. A column of $n/2$ switches is placed in the right side of the two sub-networks and their inputs are numerated from 0 to $n - 1$. Let $I(i)$ denote the $i - th$ input. The connections among the two networks and the new column of switches is done by the following algorithm:

---

for $(i = 0$ to $(n/4 - 1))$
$I(2 * i) <= S(2 * i)$
$I(2 * i + 1) <= S(2 * i + n/2)$
$I(2 * i + n/2) <= S(2 * i + 1)$
$I(2 * i + n/2 + 1) <= S(2 * i + n/2 + 1)$
end for

Finally, we connect the final wires of the butterfly PN to different output addresses to create the Banyan PN. Let $B(i)$ denote the $i - th$ output of the butterfly PN and $O(i)$ denote the $i - th$ output address of the Banyan PN's block. We make the connections using the following algorithm:

---

for $(i = 0$ to $(n/2 - 1))$
$O(i) <= B(2 * i)$
$O(i + n/2) <= B(2 * i + 1)$
end for

Figure 4.6 shows the construction of an 8x8 Banyan PN using the algorithms from above.

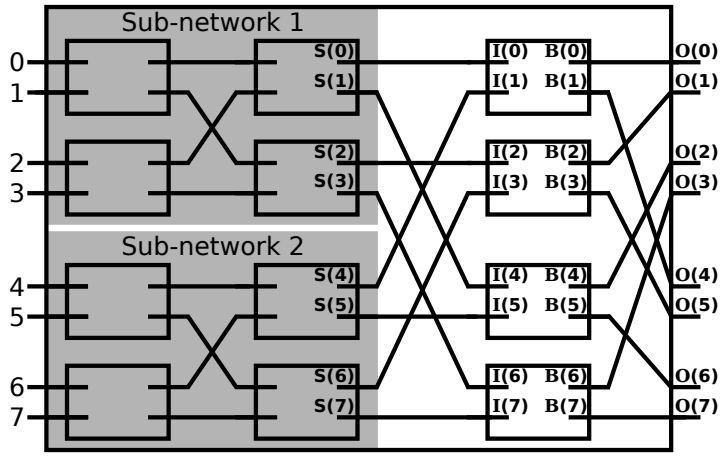

Figure 4.6: An 8x8 Banyan PN construction example.

Table 4.1: Comparison between the Benes and the Banyan PN.

Network	Possible permutations	Num switches	Stages
Benes	$n!$	$nld(n) - n/2$	$2 * ld(n) - 1$
Banyan	$n^{n/2}$	$n/2 * ld(n)$	$ld(n)$

The Banyan PN has approximately half the number of stages of the Benes network. Table 4.1 shows differences between meaningful metrics of these two networks considering n inputs. Compared with the Benes $2ld(n) - 1$ stages, Banyan network has only $ld(n)$ stages, which reduces the signal propagation time when performing the permutation. For better visualization, Figure 4.7 shows the number of switches used by these two networks as function of the number of inputs. Moreover, it is much easier to generate control signals for the Banyan PN.

The Banyan PN is non-blocking to perform all the permutations required in the trellis decoders. Non-blocking means that the network can link all the necessary paths for a desired permutation. To perform the XOR operation between the state metric index and the column vector of the PCM, the control of the Banyan PN is very simple. Each bit of the column vector controls one entire column of the PN. If the Banyan network is constructed as above, the least significant bit of the PCM's column vector controls the most left column of the network. No additional control logic is required.

4.3 Survival Memory and Traceback

After the state metrics recursion has run n times, the decision bits for all the states are stored in the survival memory. The traceback operation extracts the most likely sequence of state transitions in the Trellis. Figure 4.8 shows the architecture of the Survival Management Unit. To realize the traceback operation, we start by reading from the survival memory the decision bits from the last column of states in Trellis S_n. The first decision bit to be read in traceback is always $dec_{0,n}$, from the first state in the last trellis step, $S_{0,n}$. Therefore, the $(n - \kappa)$ bits state index register is reseted to zero and the $2^{n-\kappa}$ to one multiplexer selects the decision bit of the state metric pointed by this address. Given the decision bit from a state $S_{i,k+1}$, we can calculate the index l of the previous state in Trellis as:

$$l = i \oplus h_{k+1} * dec_{i,k+1}, \tag{4.2}$$

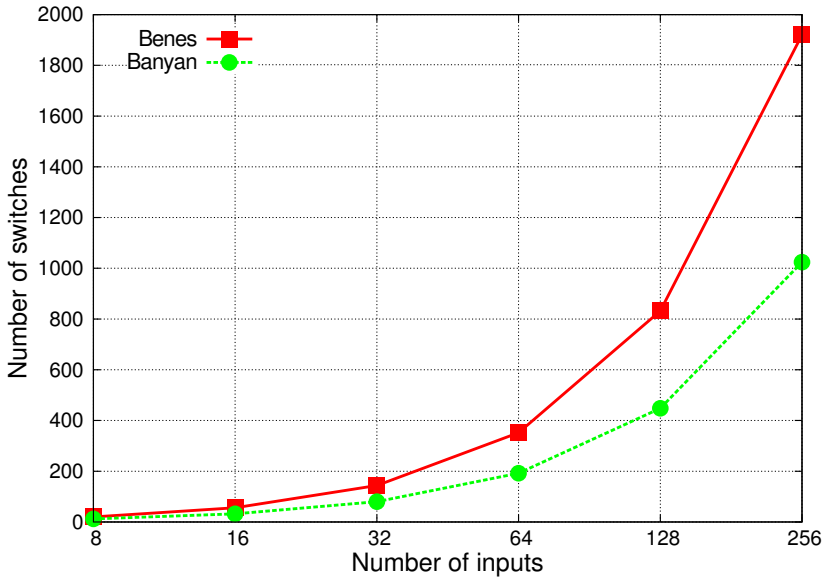

Figure 4.7: Number of switches used by the Benes and the Banyan PN

where $\oplus$ designates an xor operation. Note that information on the parity check matrix of the code is once again necessary. The index register stores the index of the previous state in Trellis and the decision bit from this state is again selected. This process continues until the first column of decision bits is read and the most likely codeword $\hat{c}$ is derived.

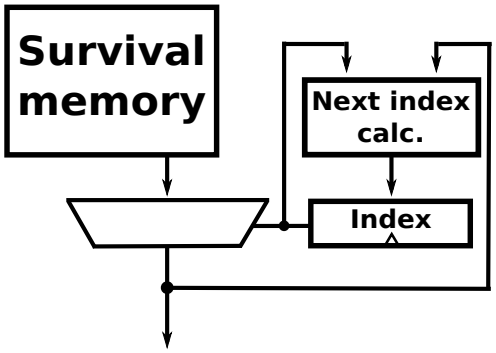

Figure 4.8: SMU architecture.

4.4 Doubling the Throughput

If only one survival memory is used, the recursion unit stays in idle mode while the traceback operation is running. The resulting throughput (decoded bits per second) of the system is then half the clock's frequency. The throughput of the decoder can be duplicated if two survival memories are used. While the recursion unit writes in one of the memories, the traceback algorithm reads the data of the second one. The memories alternate their roles every time that a new block is received. The parity check matrix ROM memory in this implementation must be dual port, since different columns of the PCM are used in

the recursion unit and in the survivor management unit. The resulting data processing of both serial and parallel decoders is depicted in Figure 4.9.

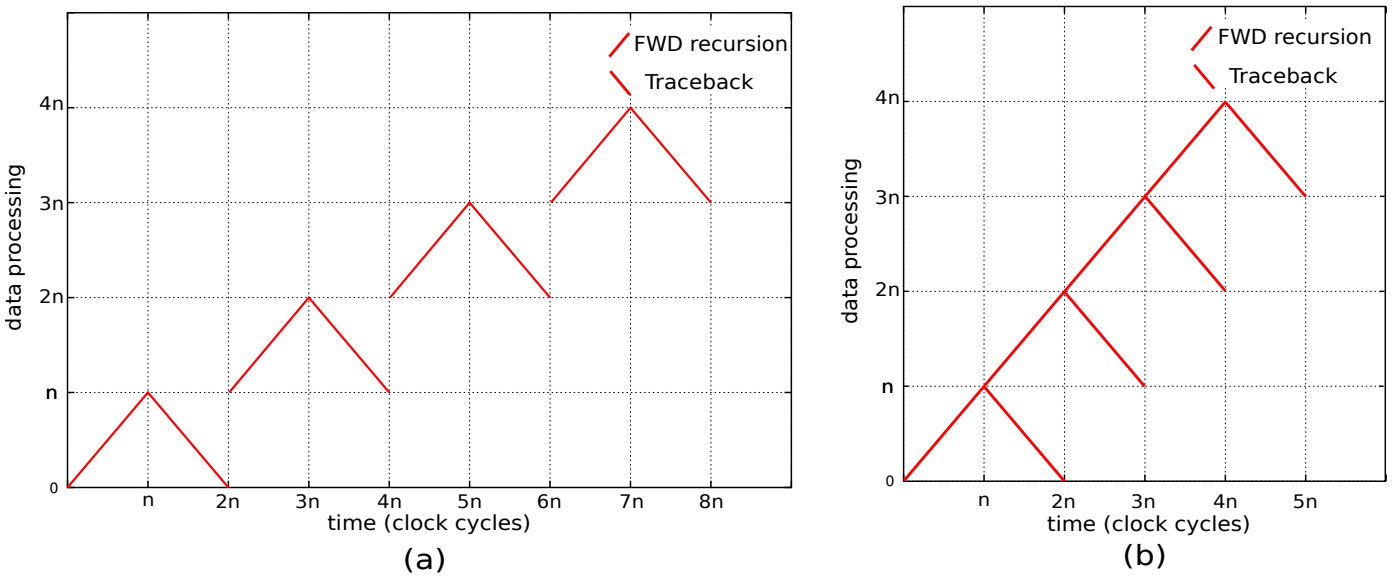

Figure 4.9: Data processing of the Viterbi decoder using: (a) one survivor memory and sequential processing (b) two survivor memories and parallel processing

One extra memory is required in this implementation, but even though it is much more efficient than the first one. Only 1% of the number of Look Up Tables (LUTs) increases due to data merging and distribution.

4.5 Quantization

Fix point representation is the best choice for implementing decoding algorithms. The floating point numbers of, e.g. the LLRs are mapped to fix point number with some rounding. Fix point arithmetic is much less complex than floating point arithmetics, but its restricted range and precision might lead to communication performance losses. The notation (q, f) is used to represent the quantization of a fix point number: q is the total number of bits and f the number of bits used in the fractional part. The precision of this representation is equal to the least positive value representable: 2^{-f}.

If two's complement arithmetic is used, the input to the decoder are numbers ranging from -2^{q-f-1} to $2^{q-f-1} - 2^{-f}$. Figure 4.10 shows the quantization process. If the value to be represented is out of this range, then we saturate the metric and the minimum or maximum value representable is used.

Input quantization.
The bit widths of the input and output LLR are chosen depending on the algorithm to be implemented. For the input of the VA a (5,1) quantization represents a slight performance loss in comparison to the floating point implementation and is the one with the best cost benefit. The plot of Figure 4.11 shows the VA Frame Error Rate (FER) for the enhanced Hamming (32,26) code as function of the Signal to Noise Ratio (SNR) considering an AWNG channel and BPSK modulation for different quantizations. The proximity between the curves is also very similar if we consider other codes.

No considerations over the best input quantization to be used for the MAX-Log-MAP

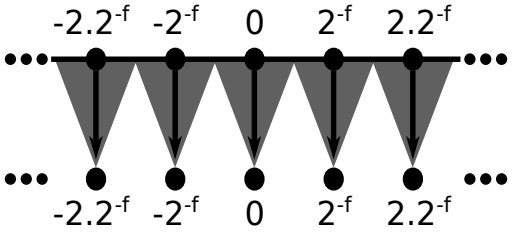

Figure 4.10: The quantization process.

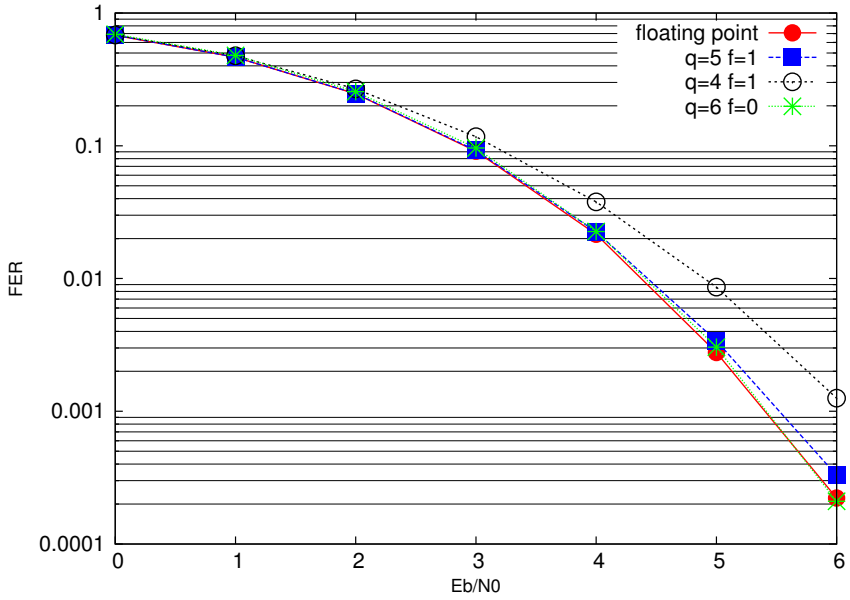

Figure 4.11: VA FER for the Extended Hamming (32,26) code considering different quantizations.

decoder shall be given here. The MAX-Log-MAP decoder is a tool used in systems that employ concatenated decoding algorithms and the best quantization depends on the system itself. The common literature usually agrees with 5-6 bits for the MAX-Log-MAP input and 6 to 8 bits for its soft output values [2].

Branch and State Metrics quantization.

For block codes, the branch metrics does not require any calculation and they are obtained directly from the input symbol LLR. Thus, the bit width of the branch metrics is the same as used for the input LLRs. For convolution codes branch metric calculation is needed, since each state transition generates two or more bits.

The calculations presented in the algorithms of Section 3 consider unnormalized state metrics, which can accumulate values without any bound. In fix point implementations, this would lead to arithmetic overflows and communication performance losses. However, some techniques might be used to avoid these overflows and they make use of two fundamental properties of Trellis based decoding algorithms [14]:

1. The maximum difference between two state metrics of one Trellis step is bounded by a fixed quantity $\Delta_{sm,max}$.

2. The difference between the state metrics is the only information relevant for the decisions of the VA and also for the soft-output calculation of the MAX-Log-MAP.

To deduce the worst case of $\Delta_{sm,max}$, we consider paths starting from the all zero state $S_{0,0}$. If the first $n-\kappa$ rows of the code's PCM are lineary independent, after exactly $n-\kappa$ (redundant bits number) stages, all the $2^{n-\kappa}$ states can be reached and there exists a path between every state $S_{i,n-\kappa}$ and $S_{0,0}$. Considering that, for block codes, the maximum difference between two branch metrics within a single Trellis step $\Delta_{\lambda,max}$ is equal to $min(\lambda)$, which is the minimum value that a branch metric may assume. The state metrics can decrease in the worse case by $\Delta_{\lambda,max}$ at each step. Thus, the maximum difference between the state metrics is given by:

$$\Delta_{sm,max} = (n-\kappa).\Delta_{\lambda,max} \tag{4.3}$$

Each extra bit used for the SM representation requires an extra PN. The bit 0 of all state metrics is routed to a permutation network, the bit 1 to another and so on. Hence, the bit width of each SM affects the total area of the circuit directly and should be carefully chosen.

4.6 Modulo Normalization

Normalization techniques are used to deal with the arithmetic overflows and to keep the combinational path delay of the ACS recursion as small as possible. In [2], three state metric normalization methods are presented. In this thesis we will consider only the modulo normalization because of its easy implementation. Moreover, the other two methods require some extra rescaling units that increase the combinational path delay of the ACS recursion. Modulo normalization does not affect the critical path.

The idea of the modulo normalization is to accommodate the overflows by employing two's complement arithmetic. The state metrics sm are mapped to its modulo metrics $\tilde{sm}$:

$$\tilde{sm} = ((sm + 2^{q-1}) mod 2^q) - 2^{q-1} \tag{4.4}$$

Instead of moving along the real line, the state metrics move around a circle with circumference 2^q. Following [2], if the difference between two metrics is bounded by a value smaller than 2^{q-1}, then their modular difference is equal to their actual difference. Thus, the decisions of the decoding algorithms are not affected.

The Viterbi decoder works properly for all codes if its state metrics quantization has the bit width necessary to represent $\Delta_{sm,max}$ plus one extra bit given the modulo normalization:

$$q_{sm} = ld(n-\kappa) + ld(\Delta_{\lambda,max}) + 1, \tag{4.5}$$

where $ld(\Delta_{\lambda,max})$ is the number of bits used for the input LLRs.

In fact, $\Delta_{sm,max}$ is a pessimistic bound and occurs rarely in block codes. We tested the results of the Viterbi decoder using different state metric quantizations and we found out that for some codes, the decoder works properly using one or two bits less than stated in Equation 4.5, even considering the worst case. So, there is clearly a code dependency on the state metrics quantization.

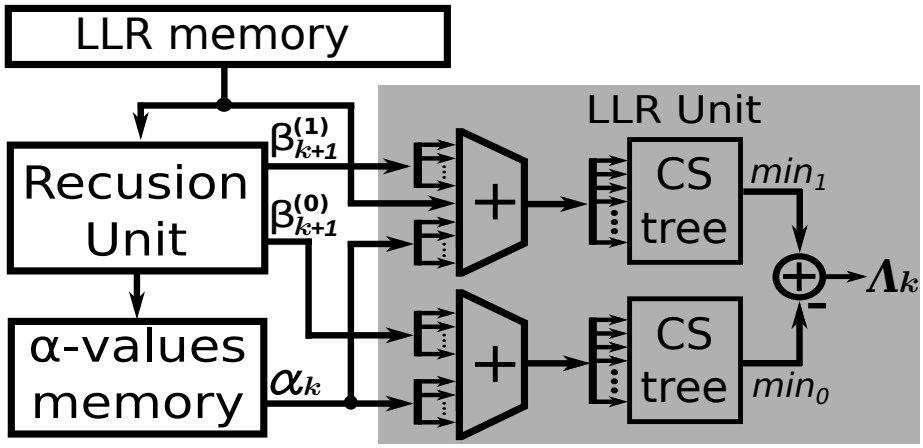

Figure 4.12: The MAX-Log-MAP decoder architecture.

The easiest way to use the state metrics for the LLR calculation of the MAX-Log-MAP is to use the same quantization for soft-output calculation and the state metrics. The common literature agrees with a bit width between 8 and 11 for the state metrics of the MAX-Log-MAP decoder, depending on the number of Trellis states and the decoding system [2].

As stated in Section 4.1 the state metrics need to be initialized before the forward and backward recursion. The metrics from the first state need to be initialized with zero and the others with ∞, as they do not exist in the first Trellis step. Since ∞ is not quantizable, the value $\Delta_{sm,max}/2$ is chosen. It represents a sufficient low probability for these metrics and does not compromise the results of both Viterbi and MAX-Log-MAP decoder.

4.7 The MAX-Log-MAP Decoder

We presented the MAX-Log-MAP algorithm in Section 3.4. We now focus on the architectural details of its hardware implementation. The recursion unit used in the Viterbi decoder is reused here.

Soft Output Decoding

The MAX-Log-MAP decoder calculates the approximated APP LLRs. We obtain the approximated probabilities from Equation 3.14 without the correction terms $f_c(|\delta_2 - \delta_1|)$ as:

$$\ln \frac{Pr(c_k = 1|r)}{Pr(c_k = 0|r)} \cong \min_{\forall(i,l)}(\gamma_{k,k+1}^{i,l}(c_k = 1) + \alpha_{i,k} + \beta_{l,k+1}) - \min_{\forall(i)}(\alpha_{i,k} + \beta_{i,k+1}). \quad (4.6)$$

The term $\gamma_{k,k+1}^{i,l}(c_k = 0)$ is omitted here, since all the branch metrics from zero-transitions are zero.

Figure 4.12 shows the basic building block of the MAX-Log-MAP architecture. Using a single recursion unit, the data gathering for the output calculation of Equation 4.6 works as follows:

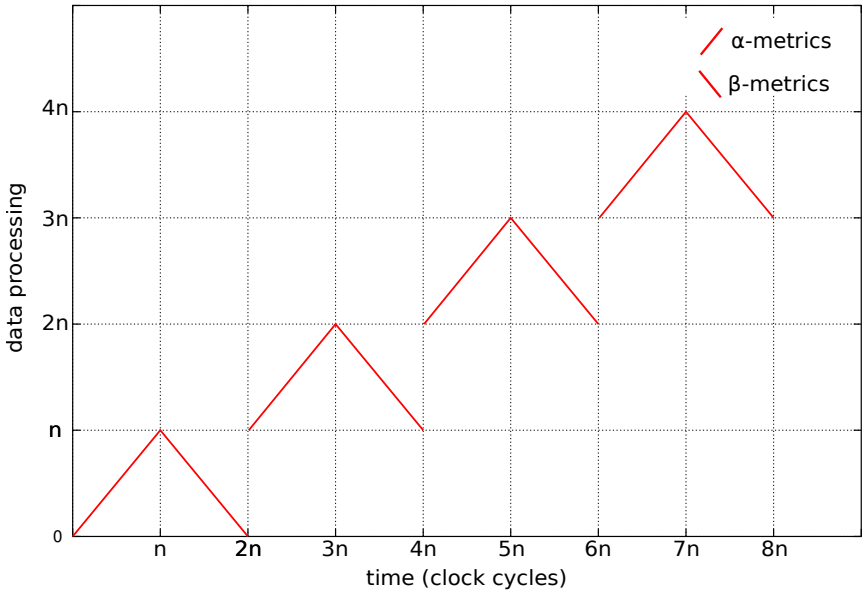

Figure 4.13: MAX-Log-MAP decoder data processing.

Both α and β-metrics are needed to compute the LLRs. Using a single recursion unit, this leads to a serial processing of state metrics as shown in Figure 4.13. The set of α-metrics calculated during the forward recursion must be stored in the α-values memory for the whole data block. During the backward recursion, all the information necessary for the computation of the LLRs is obtained. We start the computation of the LLRs from the last bit of the block. The remainder of the probabilities are calculated while the backward recursion advances.

The minimum of the sums of α, β and γ-metrics for all zero-transitions and for all one-transitions have to be found and afterwards subtracted from each other. Given the structure of the Wolf Trellis, the index of the β-metrics $\beta^{(1)}$ to be added for the calculation of the one-transitions summation are time variants. They are obtained from the output of the PN, because they are the same metrics to be compared to by the ACS units during the backward recursion. The β-metrics $\beta^{(0)}$ participating in the zero-transitions summation are gathered from the state metrics' registers. The input LLRs participate only in the one-transitions summation, since the branch metrics of all zero transitions are zero.

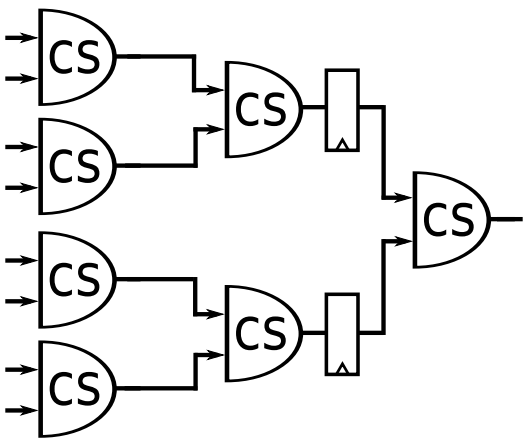

Figure 4.14: A compare select binary tree.

The α-metrics, starting with the metrics from the states S_{n-1}, are retrieved from the memory. Two blocks of adders add the α-metrics in sequence with the $\beta^{(1)}$ and $\beta^{(0)}$-metrics. As the branch metrics also participate in the one-transitions summation, the upper adders block in Figure 4.12 has $2^{n-\kappa+1}$ adders, while the lower one has $2^{n-\kappa}$ adders.

After the sums are done, a binary tree of compare select (CS) units selects the lowest values among all the one-transitions sums, min_1. Since we also have to calculate the minimum of the sums for all the zero-transitions, min_0, a second binary tree is used. Figure 4.14 shows the architecture of a CS tree with 8 inputs. Each tree is comprised of $2^{n-\kappa} - 1$ CS units, which are implemented like the compare select functions of the ACS units (a subtracter and a multiplexer). Pipeline is here considered, given the high delay of such structure.

After the lower values have been obtained from both trees, the soft-output Λ_k is calculated by subtracting min_0 from min_1.

4.8 FPGA Implementation

The architectures presented in the previous sections were described in VHDL using Xilinx ISE 14.1 release.

In order to map the above architectures for block codes of different sizes to an FPGA design we first developed a VHDL package with the code parameters and the components to be used. As to do so, it is possible to change the code to be decoded by only amending this file. The code parameters are code length, number of redundant bits, input and state metrics quantization. The components are the basic building blocks of the decoders. They include ACS units, Permutation Network, adders, Compare Select tree and the Memories.

The building blocks of the Viterbi (Figure 4.1) and MAX-Log-MAP decoder (Figure 4.12) were then implemented by instantiating the components and connecting them through buses. Not only the components itself but also the number of components to be instantiated depend on the code's parameters. Since the Permutation Network is build up in a recursive form, one file for each network of different size was constructed. The correct PN is instantiated based on the number of redundant bits of the code.

Moore Finite State Machines are used to control the operative part of the decoders. The Viterbi decoder's control is shown in Figure 4.15. Note that the Forward Recursion state runs only $(n-1)$ times in a row, instead of n times. This is because the decision bits of the last trellis step are stored in the Prepare Traceback state. In this state, the state metrics registers and the auxiliary counters are reseted. This allows a throughput of one decoded bit per clock cycle, as these decision bits are stored while the registers are configured to start a new recursion.

The controlling state machine of the MAX-Log-MAP is depicted in Figure 4.16. The α-values are calculated and stored during the forward recursion. The output calculation occurs along with the backward recursion. An auxiliary counter c is used to control the state transitions. Pipeline is used in the LLR unit after the one and zero-transitions sum and in the middle of the CS-Tree. To correctly control the circuit with pipeline two extra states are needed. In the Waiting Latency state, the backward recursion starts but the useless values generated by the LLR unit are not stored. In the beginning of the next forward recursion two correct output values in the pipeline queue need to be stored. This happens in the Forward Recursion & Output state.

Finally, the top level architectures bring these blocks and the control unit together Constructing the hardware in such hierarchical manner makes the implementation simpler

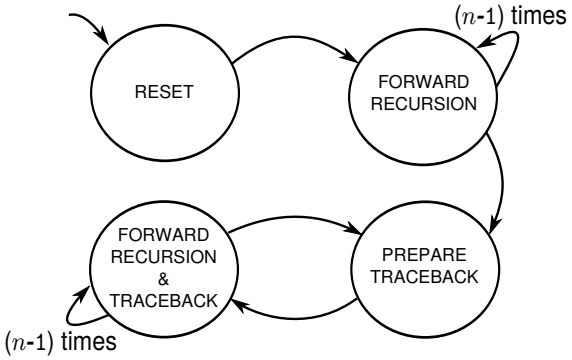

Figure 4.15: Controlling state machine of the Viterbi Decoder.

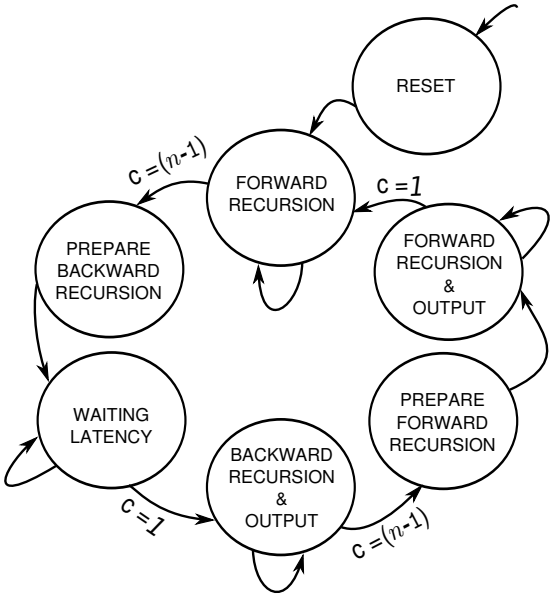

Figure 4.16: Controlling state machine of the MAX-Log-MAP Decoder.

and also easier to check the exact resources consumption of each part of the circuit. The Appendix C shows the VHDL code of the package and the two top levels developed.

5 IMPLEMENTATION RESULTS

In this section, we analyze the implementation results of the Viterbi and MAX-Log-MAP decoder architectures presented in the previous section. The circuit area and decoding throughput of both decoders are given considering FPGA implementations of different codes. We implemented the decoder architectures of Section 4 in Xilinx ISE 14.1 release using a Xilinx Virtex 6 FPGA device XC6VLX75T with speed grade -3. All the results were obtained after Place and Route.

For both decoders, we made the analysis of the results considering the decoder's throughput and area. The area analysis is divided into three components: logic area (LUTs), register used and memories.

5.1 Viterbi Decoder

Table 5.1 shows the resources (LUTs) used by the permutation network and the ACS units as well as the total number of LUTs and registers utilized in the decoder for a different number of Trellis states. The bar graph in Figure 5.1 shows the number of LUTs used as function of the number of Trellis states. The plots in this section consider a fixed code length of 255, 5 bits for the input LLRs and 7 bits for the state metrics. The x-axis is in log_2 scale, which gives an exponential curve. In the graph the area contributions of the permutation network and the ACS units are discriminated. The label "others" refers to those LUTs used by the control unit, the survivor management unit (traceback) and in additional routing paths. The resources used by both of these blocks were obtained in the detailed MAP-Report.

Analyzing the graphs, we can conclude that the decoding complexity of linear block codes exponentially depends on the number of redundant bits of the code $(n - \kappa)$. This is because during the construction of a Trellis diagram for block codes, the number bits used to represent the states is equal to the number of rows of the PCM, which gives a total of $2^{n-\kappa}$ states. The number of resources needed for decoding is proportional to the number of Trellis states.

The permutation network used to permute the state metrics data during the ACS recursion occupies a large area percentage of the circuit. The bigger the number of redundant bits of the code, the bigger the area percentage that the PN occupies (See Figure 5.1). This is because the other blocks area grows proportional to $2^{(n-\kappa)}$, while permutation network grows proportional to $(n - \kappa).2^{(n-\kappa)-1}$, which is a more than exponential increase.

In the Viterbi decoder, the number of registers used in the control unit is almost constant for all codes. Figure 5.2 shows the number of registers used in its implementation for different state metrics numbers. As the bar graphic shows, the majority of the registers are used in the state metrics.

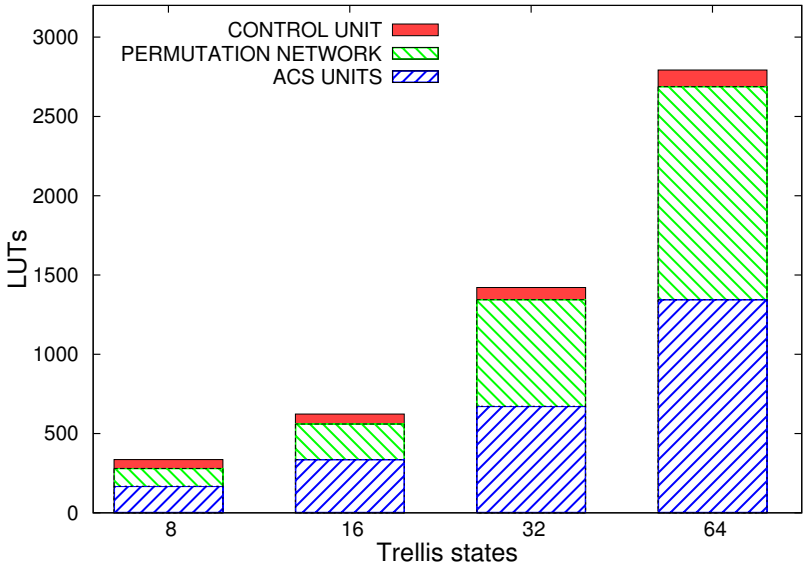

Figure 5.1: Area occupied by each component of the Viterbi decoder

Table 5.1: Resources used in the Viterbi decoder

Trellis states	ACS LUTs	PN LUTs	Total LUTs	Registers
8	124	72	265	79
16	248	192	615	144
32	496	560	1190	257
64	992	1152	2791	482

Different code lengths do not influence the throughput of the system, since the maximum combination path of this architecture is the ACS recursion unit and it is the same for different code lengths. An increase in the code length only generates an increase in the circuit's area due to the bigger survival and PCM memory.

The critical path of the Viterbi decoder is reading from the parity check matrix ROM memory, permuting the state metrics data in the permutation network, calculating the next metrics in the ACS units and storing the decision bits in the survivor memory. Figure 5.3 shows the number of decoded bits per second as function of the number of Trellis states. Since we did not implement pipeline in the recursion unit and the number of stages of the Banyan PN increases by one every time we double the number of its inputs, the maximum combination path delay of the RU increases linearly with the number of redundant bits, given the extra stage in the permutation network needed, resulting in a lower throughput.

The Viterbi decoder uses three memories: a parity check matrix memory and two survivor memories. The memories are mapped to 36kbit and 18kbit block RAM memories as displayed in Table 5.3. The size of the memories is proportional to the code's length. Table 5.2 shows the size in bits and the type of the memories used for different trellis states numbers considering a code length of 255. Two survivor memories are used due to the throughput problem discussed in Section 4.4. These two memories store the decision bits for all the state metrics during the forward recursion. Thus, they grow proportional to the number of state metrics.

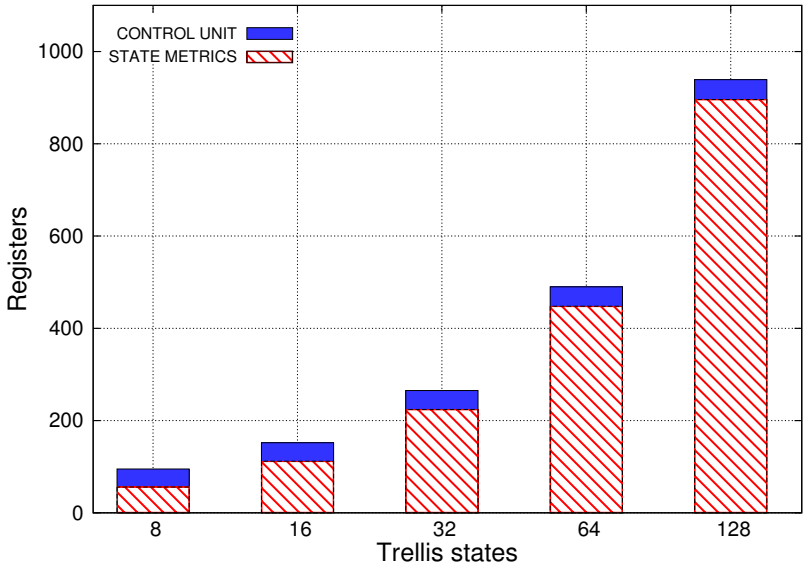

Figure 5.2: Viterbi decoder's registers as function of the number of Trellis states.

Table 5.2: Memories used in the Viterbi decoder

Trellis states	Survivor memories	PCM memory
8	2 255x8-bit single port RAM	255x3-bit dual port ROM
16	2 255x16-bit single port RAM	255x4-bit dual port ROM
32	2 255x32-bit single port RAM	255x5-bit dual port ROM
64	2 255x64-bit single port RAM	255x6-bit dual port ROM

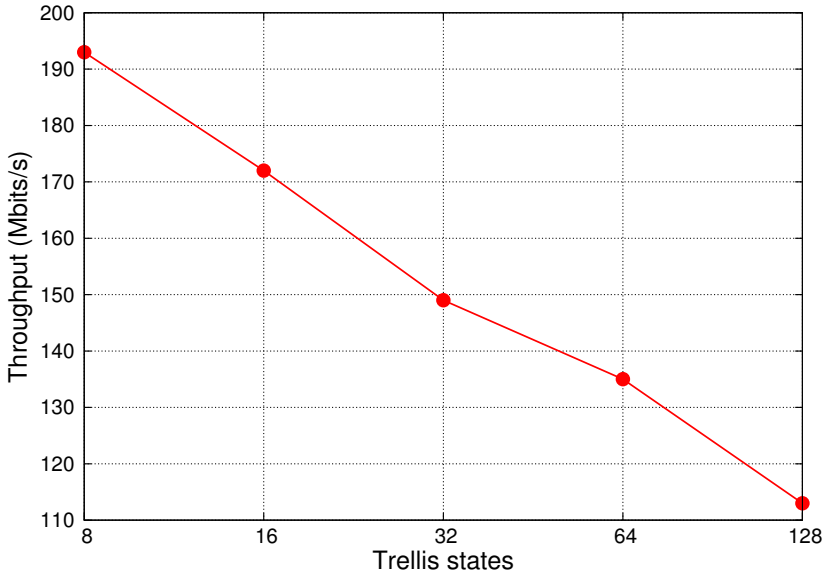

Figure 5.3: Viterbi decoder throughput as function of the number of Trellis states.

Table 5.3: Memory mapping Viterbi decoder

Trellis States	RAMB36E1	RAMB18E1
8	0	3
16	0	3
32	0	3
64	2	1
128	4	1

5.2 MAX-Log-MAP Decoder

The area contribution of each block of the circuit is displayed in Table 5.4.

The total area (LUTs) of the decoder as well as the area contribution of each block that the decoder comprises is plotted in the bar graph in Figure 5.4 for different Trellis States (TS) numbers. The "others" label refers to the LUTs used by the control unit and in additional routing. We obtained the data in all graphics presented in this section using a fixed code length of 255, 6 bits for the input LLRs and 8 bits for the state metrics and also for the LLR calculation.

The total circuit's area of the MAX-Log-MAP decoder also increases exponentially with the number of redundant bits. The area of the MAX-Log-MAP decoder is bigger than the Viterbi decoder's, given the extra resources (adders and CS trees) used for the LLR calculation in the LLR unit. The LLR unit occupies approximately 50% of the decoder's area, and is surely the unit that we have to look into with more detail if we want to improve the area consumption.

In this architecture, a single recursion unit calculates the α and β-metrics in a forward and backward recursion consecutively. Thus, the real throughput of the decoder is half the circuit's clock frequency. Since pipeline is used in the LLR unit of the MAX-Log-MAP, its critical path is the ACS recursion. The max combinational path in the RU comprises: reading from the PCM memory, permuting the state metrics data in the permutation network, calculating the next metrics in the ACS units and writing in state metrics' registers. If two recursion units are used, one decoded bit per clock cycle can be achieved with costs of a larger area. The graphic in Figure 5.6 shows the decoder's throughput for the implemented design. Also, because one extra stage in the PN is needed every time a redundant bit is added, the throughput of the decoder decreases linearly with the number of parity bits.

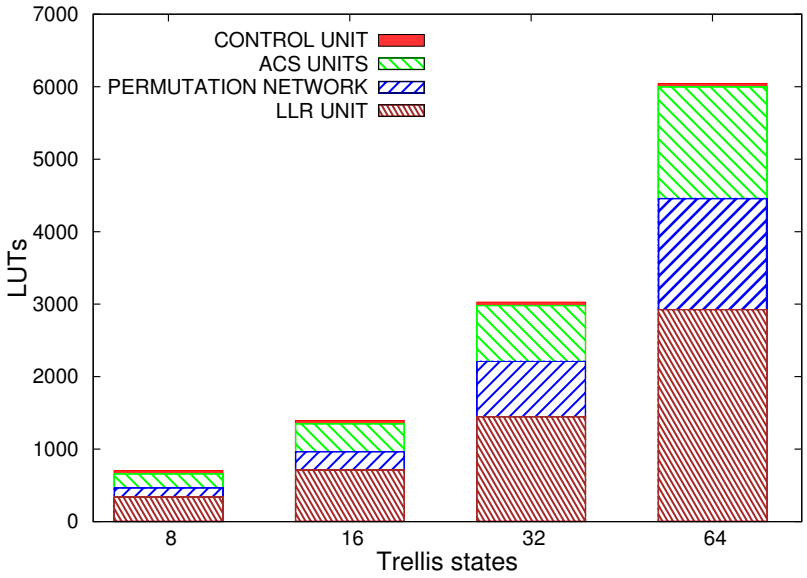

Figure 5.4: MAX-Log-MAP decoder LUTs usage as function of the number of Trellis states.

Figure 5.5 illustrated the number of registers used in the MAX-Log-MAP decoder for different state metrics numbers. The number of registers used in the control unit is constant for all the codes. The MAX-Log-MAP decoder has 2 pipeline stages, both in the LLR unit: one after the one and zero transitions summation and the other in the middle

Table 5.4: Resources used in the MAX-Log-MAP decoder

TS	ACS LUTs	PN LUTs	LLR Unit	Total LUTs	Registers
8	160	96	331	733	255
16	320	256	821	1523	447
32	640	896	1541	3166	863
64	1280	1536	3265	6852	1639

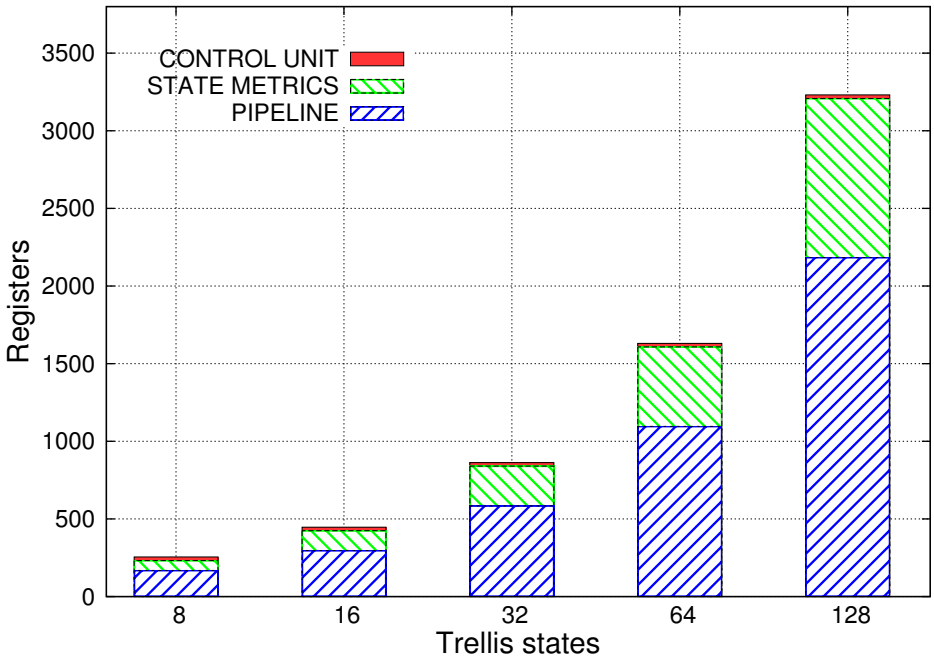

Figure 5.5: MAX-Log-MAP decoder's registers as function of the number of Trellis states.

of the compare select trees. Pipeline is needed here because of the high delay of this unit. The number of state metrics registers is also plotted.

Three memories are used in the MAX-Log-MAP decoder: an α-values memory, a LLR memory, and a PCM memory. Table 5.5 shows the number of bits and the type of each memory for different trellis states numbers, considering codes with 255 bits length. The memories are mapped to 36kbit and 18kbit block RAM memories as displayed in Table 5.6. In the MAX-Log-MAP decoder, the size of the memories is also proportional to the code's length. The memory that grows faster is the α-values memory, since it has to store all the α-metric's data before the backward recursion begins. The LLR memory has a fixed size, due to the that fact that the input quantization is the same for all the codes. An LLR memory is used in the MAX-Log-MAP decoder, because the input values need to be retrieved in the backward recursion.

Table 5.5: Memories used in the Max-Log-MAP decoder

TS	α-values memory	LLR memory	PCM memory
8	255x64-bit single port RAM	255x6-bit single port RAM	255x3-bit single port ROM
16	255x128-bit single port RAM	255x6-bit single port RAM	255x4-bit single port ROM
32	255x256-bit single port RAM	255x6-bit single port RAM	255x5-bit single port ROM
64	255x512-bit single port RAM	255x6-bit single port RAM	255x6-bit single port ROM

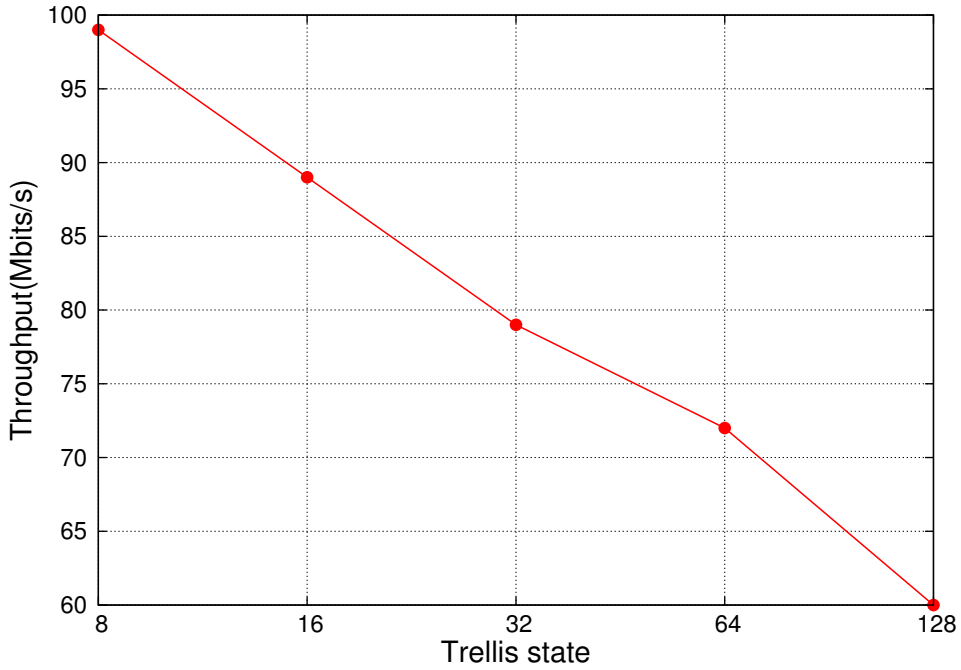

Figure 5.6: MAX-Log-MAP decoder throughput as function of the number of Trellis states.

Table 5.6: Memory mapping Max-Log-MAP decoder

Trellis States	RAMB36E1	RAMB18E1
8	1	2
16	2	2
32	4	2
64	8	2
128	16	2

5.3 Validation of the Work

It is clear that the results generated by the hardware architectures presented in this thesis need to be validated. In this section we give a quick overview on how we tested and validated our hardware design.

We started by developing software programs for both algorithms discussed in this thesis (the MAP and the VA) which used floating point arithmetics and a high abstraction level. The frame error rates generated by the VA software were compared with the frame error rates of well known block codes (Hamming and BCH codes) considering ML decoding. The soft-results of the Max-Log-Map also needed to be validated. We tested the results of the Max-Log-Map by comparing them to the VA software's outputs. If both algorithms have the same input symbols, than the hard decisions of the Max-Log-Map algorithm's output should be equal to the VA's output.

The next step was to reduce the complexity of the floating point arithmetic used. Thus we developed new software programs for both algorithms which used only bit vectors and quantization. These softwares are a model for the hardware to be implemented. The error rates of the Viterbi decoder's software model were plotted along with the error rates of the software that uses floating point. An example of such plot can be seen in Figure 4.11. Due to the roundings caused by the quantization process, little losses in communication performance are expected. The results of the Max-Log-Map were simply compared with the results of the floating point MAP software. Our goal was to look for big deviations from the expected values, since small deviations are expected due to the quantization. We also tested these lower level softwares considering worst input cases (LLRs with high absolute values), in which arithmetic overflows would be more likely to happen.

Finally, we implemented both hardwares in VHDL. The hardware designs were tested by comparing their results with the results of the software models. The tests were made for different codes with code lengths from 7 to 255 and number of parity bits from 3 to 8. We applied over 100 000 tests for each code tested and all the results were equal to those generated by the software models.

6 CONCLUSION

In this thesis, we have investigated the topic of Hardware implementations of Trellis based decoders for Linear Block codes. The biggest problem in constructing Trellis based decoders hardware for block codes was the trellis time variance. A solution for this was found by using a Banyan permutation network.

In the course of our investigation we encountered advantages and disadvantages in the decoding approach used. The important advantage of our task is that the decoders' complexity does not depend on the code length. For bigger codes, the recursions will take more time as more steps for this are needed and also bigger memories are needed. Furthermore, in a Trellis for block codes, each stage comprises only one bit, which eliminates the necessity of calculating branch metrics.

In comparison to that, disadvantages that cannot be set aside were that a Trellis for block codes is time variant and a permutation network needs to be used to arrange the state metrics data before the recursions proceed. Moreover, the decoders' areas increase exponentially with the number of parity bits for the code.

Taking into account the advantages and disadvantages of our Trellis based decoders, we can say that this application is very suitable for high code rate codes, since they usually have high code lengths and the costs of adding parity bits to the codes to be decoded are high.

It is important to say that the contents presented in this work have been published at the Advances in Radio Sciences Journal (September/2013). The article we have submitted is shown in Appendix B.1.

6.1 Future Work

In conclusion, we can say that this research has proven to be very valuable for the field of ML block codes decoding. The research in this area can still be extended. This thesis can, nevertheless, provide a good basis for future work in the area of Turbo decoders for linear block codes.

The MAX-Log-MAP decoder is a tool that can be used in concatenated decoding systems to improve the decoding performance.

Turbo codes are high performance forward error correcting codes that use MAX-Log-MAP decoders and feedbacks, so the soft-outputs of the decoders can be used to build an iterative system.

If the number of redundant bits of the code is higher than 7, then a Trellis with more than 128 states is required. The state metrics can be processed in partial parallel implementations if the throughput requirements do not demand for processing all the metrics in parallel. Advantages of partial parallel processing are less area and power, and a smaller

permutation network. How to deal with the Trellis time variance for block codes and the permutation network is the biggest challenge of this task.

REFERENCES

[1] J. Wolf. Efficient maximum likelihood decoding of block codes using a trellis. *IEEE Transactions on Information Theory, Vol. IT-20, NO. 1*, January 1978.

[2] T. Vogt. *A Reconfigurable Application-specific Instruction-Set Processor for Trellis based Channel Decoding*. PhD thesis, Technische Universität Kaiserslautern, 2008.

[3] A. Viterbi. Error bounds for convolutional codes and an asymptotically optimal decoding algorithm. *IEEE Trans. Information Theory*, IT-13:260–269, April 1967.

[4] F. Jelinek L.R. Bahl, J. Cocke and J. Raviv. Optimal decoding of linear codes for minimizing symbol error rate. *IEEE Trans. on Inform., Vol. IT-20.*, pages 284–287, March 1974.

[5] Ramesh Pyndiah, Alain Glavieux, Annie Picart, and Sylvie Jacq. Near optimum decoding of product codes. In *Proceedings IEEE Global Telecommunications Conference*, pages 339–343, San Fransisco, CA, USA, November – December 1994. IEEE.

[6] Robert Michael Tanner. A recursive approach to low complexity codes. *IEEE Transactions on Information Theory*, 27(5):533–547, 1981.

[7] Liu and Lin. Turbo encoding and decoding of reed-solomon codes through binary decomposition and self-concatenation. *IEEETCOMM: IEEE Transactions on Communications*, 52, 2004.

[8] Alexander Vardy and Yair Be'ery. Bit-level soft-decision decoding of reed-solomon codes. *IEEE Transactions on Communications*, 39(3):440–444, 1991.

[9] Claude Berrou and Alain Glavieux. Near optimum error-correcting coding and decoding: Turbo Codes. *IEEE Transactions on Communications*, 44(10):1261–1271, October 1996.

[10] Todd K. Moon. *Error correction coding: mathematical methods and algorithms*. Wiley-Interscience, pub-WILEY-INTERSCIENCE:adr, 2005.

[11] Brack T. *Application and Standard Driven LDPC Code Decoder Development*. PhD thesis, Technische Universität Kaiserslautern, 2007.

[12] V. E. Benes. Optimal rearrangeable multistage connecting networks. *The Bell System Technical Journal*, 4, 1964.

[13] L. Rodney Goke and G. Jack Lipovski. Banyan networks for partitioning multiprocessor systems. *1st Annual Symposium on Computer Architecture*, pages 21–28, 1973.

[14] Andries P. Hekstra. An alternative to metric rescaling in viterbi decoders. *IEEE Transactions on Communications*, 37(11):1220–1222, 1989.

APPENDIX A ALGORITHMS EXAMPLES

A.1 An Example of Viterbi Algorithm for Block Codes

The Viterbi algorithm applied to block codes was presented in Section 3.3.2. In this appendix we give an example of encoding and decoding of a block code using the VA for a better understanding.

Consider a block code with generator matrix G and parity check matrix H as below:

$$G = \begin{bmatrix} 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \end{bmatrix} \; H = \begin{bmatrix} 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \end{bmatrix} \tag{A.1}$$

The message $m = \{1\ 0\ 1\}$ to be sent is encoded by the channel encoder through the multiplication of m with the matrix G as in Equation 2.5. The result is a codeword $c = \{1\ 0\ 1\ 1\ 0\}$, which is modulated by an BPSK modulator and transmitted into an AWGN channel. The modulated symbols are affected by random noise during the transmission.

Viterbi Decoding

The digital demodulator delivers the calculated LLRs $r = \{-2.1\ \ 1.3\ \ 0.5\ \ -1\ \ 2.3\}$ to the Viterbi decoder. Note that the hard-decisions of these symbols do not correspond to a valid codeword.

The Viterbi algorithm using a Wolf trellis is as shown in Figure A.1. The calculation of a branch metric $\gamma_{k,k+1}^{i,l}$ in the Wolf trellis is simply zero for zero-transitions ($i = l$) and the current input symbol r_k for one-transitions ($i \neq l$). $\gamma_{k,k+1}^{i,l}$ denotes the unique branch metric from a state $S_k^i$to a state S_{k+1}^l.

The state metrics of the first trellis step have to be initialized. The metric from the state zero is initialized with zero and the other ones with ∞. In the forward recursion, at each time step k, the new state metrics α_{k+1} and decision bits dec_{k+1} have to be calculated based on the previous state metrics α_k and the branch metrics as in Equation 3.11:

$$\alpha_{l,k+1} = \min(\alpha_{l,k}, \alpha_{i,k} + \gamma_{k,k+1}^{i,l}) \tag{A.2}$$

Every time a path from a zero-transition is chosen, the decision bit 0 is selected. If both paths metrics have the same value, than the path from the zero-transition is also selected. The paths which did not survive in the example of Figure A.1 are drawn in light gray. The decision bits for each node are depicted under each state metric.

During the traceback operation, the decision bits are read from the survival memory, beginning with the bits from the last trellis step. An index register contains the index

of the next bit to be read. Every time a decision bit zero is read, the value in the index register is maintained. Otherwise the index of the next bit has to be calculated by realizing an XOR operation between the current index and the column of the parity check matrix. Once the traceback has finished, the bits read during this operation compose the most likely codeword sent.

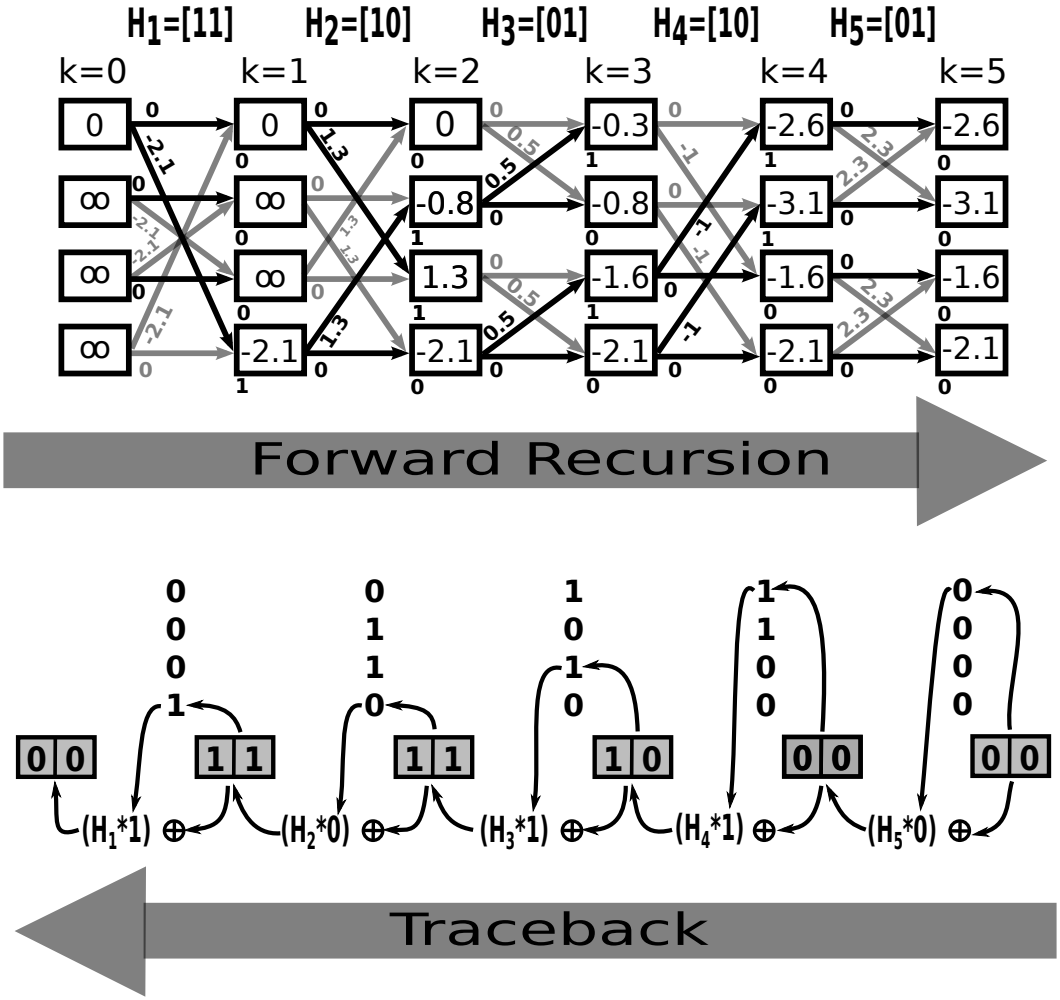

Figure A.1: A Viterbi Algorithm for Block Codes Example.

A.2 An Example of Max-Log-Map Algorithm for Block Codes

The Max-Log-Map algorithm applied to block codes was presented in Section 3.4. In this appendix we give an example of decoding of a block code using the Max-Log-Map algorithm for a better understanding.

Consider a block code with parity check matrix H as below:

$$H = \begin{bmatrix} 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \end{bmatrix} \tag{A.3}$$

A codeword $c = \{1\ 0\ 1\ 1\ 0\}$ of this code is modulated by an PBSK modulator and sent through an AWGN channel. After transmission, the corespondent input LLRs $r = \{1\ \ 2.1\ \ -2.5\ \ -2.8\ \ 0.8\} = \{r_0\ r_1\ r_2\ r_3\ r_4\}$ are delivered to the decoder.

Soft Output Calculation

The Max-Log-Map decoder's soft-outputs calculation is comprised of 3 parts: the forward recursion, the backward recursion and the calculation of the output symbols. The α-metrics are calculated during the forward recursion. They are the same metrics calculated during the VA's ACS recursion (Equation 3.11). The β-metrics are calculated in a similar way, but beginning with the last stage in the trellis diagram (Equation 3.15). Both recursions accumulate probabilities for each state based on the metrics previously calculated and the input symbols. The calculation of a branch metric $\gamma_{k,k+1}^{i,l}$ in the Wolf trellis is simply zero for zero-transitions ($i = l$) and the current input symbol r_k for one-transitions ($i \neq l$). Figure A.2 shows the calculation of the beta and alpha metrics considering a code with parity check matrix H and the LLRs above.

After the β-metrics of a trellis step have been calculated, it is possible to calculate the corespondent output symbol. To calculate an output LLR, the minimum of the sums for all one-transitions and for all zero-transitions sums have to be found and afterwards subtracted from each other. The minimum one and zero-transitions branches in Figure A.2 are drawn with dashes to facilitate their identification. The calculation of the output symbols are depicted in the equations below:

$$\begin{aligned} \Lambda_4 &= \min(\alpha_{0,4} + r_4 + \beta_{1,5},\ \alpha_{1,4} + r_4 + \beta_{0,5},\ \alpha_{2,4} + r_4 + \beta_{3,5},\ \alpha_{3,4} + r_4 + \beta_{2,5}) \\ &- \min(\alpha_{0,4} + \beta_{0,5},\ \alpha_{1,4} + \beta_{1,5},\ \alpha_{2,4} + \beta_{2,5},\ \alpha_{3,4} + \beta_{3,5}) = -2.4 - (-4.3) = 1.9 \end{aligned}$$

$$\begin{aligned} \Lambda_3 &= \min(\alpha_{0,3} + r_3 + \beta_{2,4},\ \alpha_{1,3} + r_3 + \beta_{3,4},\ \alpha_{2,3} + r_3 + \beta_{0,4},\ \alpha_{3,3} + r_3 + \beta_{1,4}) \\ &- \min(\alpha_{0,3} + \beta_{0,4},\ \alpha_{1,3} + \beta_{1,4},\ \alpha_{2,3} + \beta_{2,4},\ \alpha_{3,3} + \beta_{3,4}) = -4.3 - (-1.7) = -2.6 \end{aligned}$$

$$\begin{aligned} \Lambda_2 &= \min(\alpha_{0,2} + r_2 + \beta_{1,3},\ \alpha_{1,2} + r_2 + \beta_{0,3},\ \alpha_{2,2} + r_2 + \beta_{3,3},\ \alpha_{3,2} + r_2 + \beta_{2,3}) \\ &- \min(\alpha_{0,2} + \beta_{0,3},\ \alpha_{1,2} + \beta_{1,3},\ \alpha_{2,2} + \beta_{2,3},\ \alpha_{3,2} + \beta_{3,3}) = -4.3 - (-1) = -3.3 \end{aligned}$$

$$\begin{aligned} \Lambda_1 &= \min(\alpha_{0,1} + r_1 + \beta_{2,2},\ \alpha_{1,1} + r_1 + \beta_{3,2},\ \alpha_{2,1} + r_1 + \beta_{0,2},\ \alpha_{3,1} + r_1 + \beta_{1,2}) \\ &- \min(\alpha_{0,1} + \beta_{0,2},\ \alpha_{1,1} + \beta_{1,2},\ \alpha_{2,1} + \beta_{2,2},\ \alpha_{3,1} + \beta_{3,2}) = -2.4 - (4.3) = 1.9 \end{aligned}$$

$$\begin{aligned} \Lambda_0 &= \min(\alpha_{0,0} + r_0 + \beta_{3,1},\ \alpha_{1,0} + r_0 + \beta_{2,1},\ \alpha_{2,0} + r_0 + \beta_{1,1},\ \alpha_{3,0} + r_0 + \beta_{0,1}) \\ &- \min(\alpha_{0,0} + \beta_{0,1},\ \alpha_{1,0} + \beta_{1,1},\ \alpha_{2,0} + \beta_{2,1},\ \alpha_{3,0} + \beta_{3,1}) = -4.3 - (2.4) = -1.9 \end{aligned}$$

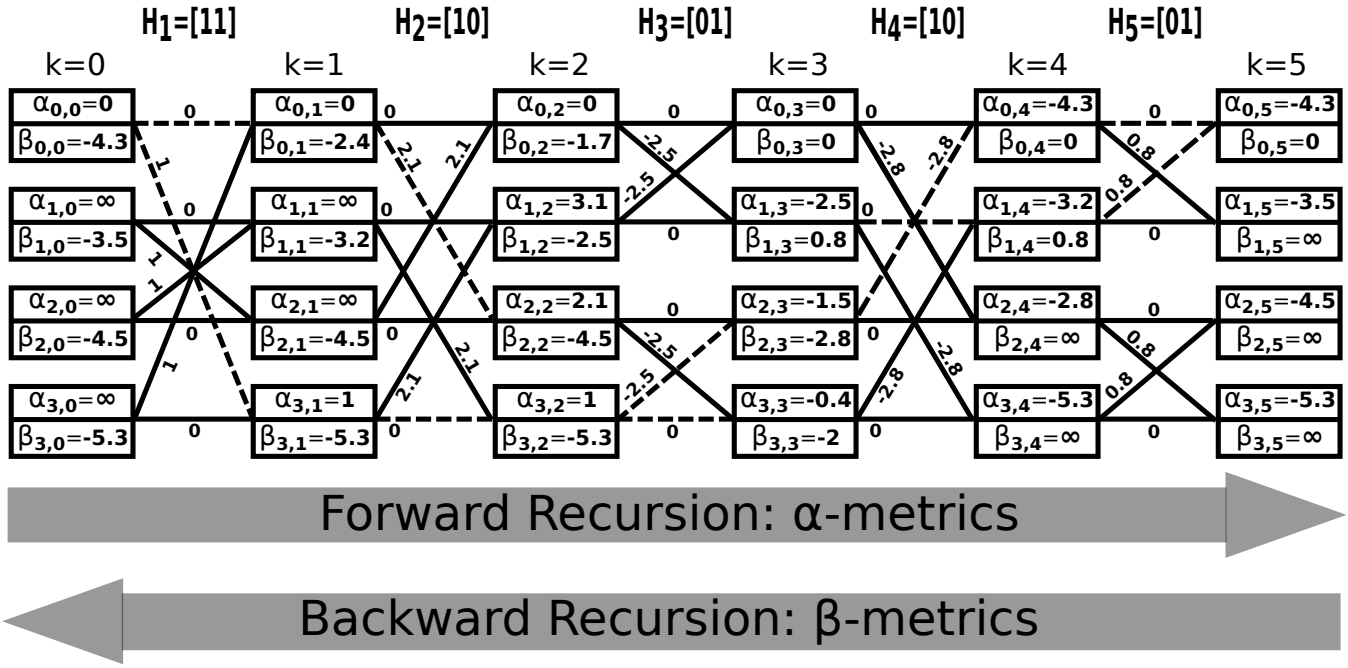

Figure A.2: A Max-Log-Map Algorithm for Block Codes Example.

Note that because the branch metrics of all the zero transitions are zero, the input symbols contribute only to the one-transitions summation. The soft-output symbols calculated by the Max-Log-Map decoder for the LLRs $r = \{1 \;\; 2.1 \;\; -2.5 \;\; -2.8 \;\; 0.8\}$ are $\Lambda = \{-1.9 \;\; 1.9 \;\; -3.3 \;\; -2.6 \;\; 1.9\}$.

A.3 A Convolutional Code Example

Convolutional codes were presented in Section 2.3. This appendix gives an example of encoding and decoding of a convolutional codes for a better understanding.

Encoding of a Convolutional Code.

The information sequence $u = \{1\ 1\ 0\ 1\ 0\ 0\}$ is encoded by the convolutional encoder of Figure 2.2 producing the encoded sequence $v = \{(1\ 1)\ (0\ 1)\ (0\ 1)\ (1\ 0)\ (1\ 0)\ (1\ 1)\}$. Note that this example already considers tail biting. It is easier to see the encoding process by looking the Mealy State Machine of Figure 2.3. Considering a BPSK modulated, the encoded bits $\{0\ 1\}$ are mapped to discrete symbols $\{-1 + 1\}$ before transmission.

Viterbi Decoding

The received sequence considering hard-decisions $r = \{(-1 - 1)\ (+1 - 1)\ (+1 - 1)\ (-1 + 1)\ (\mathbf{+1} + 1)\ (-1 - 1)\}$ includes one corrupted bit. During the ACS recursion of the Viterbi decoder, the branch metrics of each state transition in the trellis have to be calculated based on the state transitions' outputs v_k and the received symbols r_k as:

$$\gamma_{k,k+1}^{m,m'} = \sum_{\kappa=0}^{n-1} v_\kappa r_\kappa. \tag{A.4}$$

Note that $\gamma_{k,k+1}^{m,m'}$ denotes the unique branch metric for a transition from state S_k^m to state $S_{k+1}^{m'}$. As an example of branch metric calculation, consider a 1-transition from state S_k^0 to state S_{k+1}^2 in the state machine of Figure 2.3. The output v_k of this transition is $(1\ 1)$. If the received symbols at time step k is $r_k = (-1\ -1)$, than the calculation of the metric $\gamma_{k,k+1}^{0,2}$ is as follows:

$$\gamma_{k,k+1}^{0,2} = \sum_{\kappa=0}^{1} v_\kappa r_\kappa = (1\ 1) \cdot (-1\ -1) = -2. \tag{A.5}$$

The state metrics of the first trellis step have to be initialized. The metric from the state zero is initialized with zero and the other ones with ∞. In the forward recursion, at each time step k, the new state metrics α_{k+1} and decision bits dec_{k+1} have to be calculated based on the previous state metrics α_k and the current input symbol r_k.

$$\alpha_{k+1}^{m'} = \min_{\forall m}(\alpha_k^m + \gamma_{k,k+1}^{m,m'}) \tag{A.6}$$

Figure A.3 shows the calculation of state metrics and decision bits for the received symbols r above. The paths which did not survive are drawn in light gray. The decision bits have to be store in a survival memory. Every time a path from a state metric with lower state number is chosen, the decision bit 0 is selected. If both paths metrics have the same value, 0 is also selected. The decision bits for each state are depicted under each state metric.

During the traceback operation, the decision bits are read from the survival memory, beginning with the bits from the last trellis step. The shift register is reseted to all-zero and act as a pointer to the correct bit to be read. The bit recently read enters the shift register and the bits leaving the register are the most likely bits sent. Once the traceback operation has finished, the most likely path through the trellis is identified.

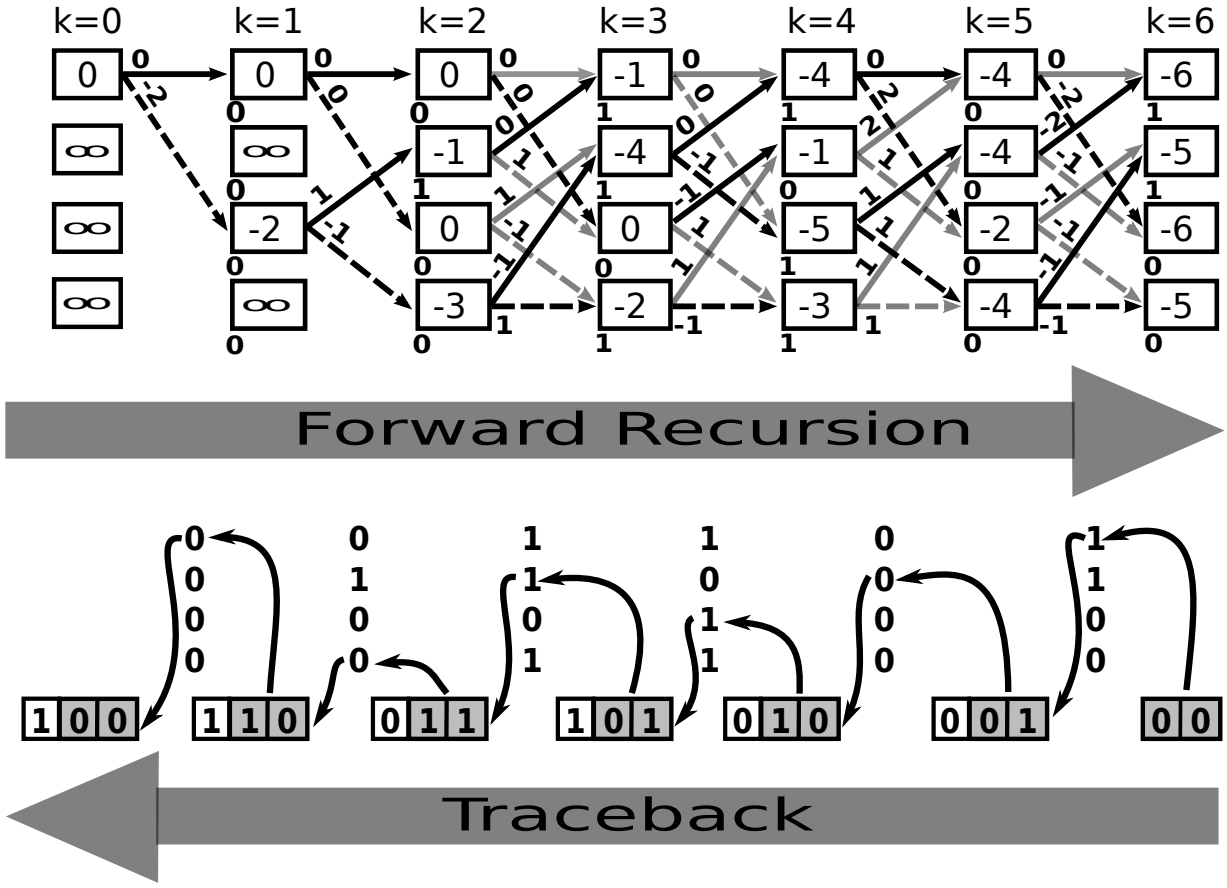

Figure A.3: A convolutional decoder example.

APPENDIX B RELATED WORK

B.1 Article published at the Advances in Radio Sciences Journal

Manuscript prepared for J. Name
with version 5.0 of the LaTeX class copernicus.cls.
Date: 13 December 2013

FPGA Implementation of Trellis Decoders for Linear Block Codes

S. Scholl[1], **E. Leonardi**[2], **and N. Wehn**[1]

[1]Microelectronic Systems Design Research Group, University of Kaiserslautern, 67663 Kaiserslautern, Germany
[2]Institute of Informatics, Federal University of Rio Grande do Sul, 91501970 Porto Alegre, Brazil

Correspondence to: S. Scholl (scholl@eit.uni-kl.de)

Abstract. Forward error correction based on trellises has been widely adopted for convolutional codes. Because of their efficiency, they have also gained a lot of interest from a theoretic and algorithm point of view for the decoding of block codes. In this paper we present for the first time hardware architectures and implementations for trellis decoding of block codes. A key feature is the use of a sophisticated permutation network, the Banyan network, to implement the time varying structure of the trellis. We have implemented the Viterbi and the max-log-MAP algorithm in different folded versions on a Xilinx Virtex 6 FPGA.

1 Introduction

Forward error correction is widely used in today's communication systems for the correction of transmission errors. In the last years and decades many different error correction schemes have been introduced and successfully adopted in various communication standards. Prominent examples for channel codes are convolutional codes, Reed-Solomon codes, turbo codes and LDPC codes.

The optimal correction strategy is called maximum likelihood (ML) decoding. Since ML decoding is very complex for many practically used codes, most of the decoding algorithms are suboptimal heuristics, e.g. the turbo decoding algorithm (Lin and Jr (2004)). However, in this paper we consider two algorithms, that can efficiently perform ML decoding of convolutional codes or small block codes: the Viterbi algorithm (VA) (Viterbi (1967)) and the BCJR algorithm (Bahl et al. (1974)).

The efficiency of the VA and BCJR algorithm originate mainly in the exploitation of the code's structure, which is graphically represented as a trellis diagram. In the past a vast amount of research has been carried out on trellis based decoding for convolutional codes, including works from general theory to hardware implementations for real world applications.

However, not only convolutional codes can be described by trellis diagrams. Also block codes can be represented as trellis and thus efficient decoding algorithms for convolutional trellises can also be applied to block codes. Trellis decoders for block codes have many different applications as standalone ML decoder or as components of larger decoding heuristics. Here we want to point out just a few use cases:

- as a maximum likelihood decoder for small block codes
- as a component decoder for turbo product codes, Pyndiah et al. (1994)
- as a check node decoder for generalized LDPC codes, Tanner (1981)
- as a component for soft decision decoding of Reed-Solomon codes, e.g. in Vardy and Be'ery (1991), Liu and Lin (2004)

Trellises of block codes mostly have a special structure (called time varying trellis), that poses a major challenge for the hardware designer. So far hardware architectures and implementation have not been considered yet. In Kim et al. (2003) a trellis decoder was implemented on an FPGA, but its use is restricted to a small group of block codes, that do not have a time varying structure.

In this paper we propose an architecture, that is able to handle all block codes of reasonable size. We solve the challenge posed by the time varying structure by introducing a optimized Banyan permutation network, that is tailored to the application. We evaluate the architectures for the VA and BCJR as well as folded versions by implementing them on a Virtex 6. To our best knowledge, this is the first hardware implementation of trellis decoding for arbitrary block codes.

The paper is structured as follows: In Section 2 we first present the construction rules for a trellis, followed by a

brief description of the algorithms in Section 3. Section 4 describes the proposed architectures and the implementation results can be found in Section 5.

2 Trellis Construction for Block Codes

We consider a binary block code with block length N and K information bits. The code is defined by its parity check matrix (PCM) $\mathbf{H}$ of dimension $(N-K) \times N$. The columns of $\mathbf{H}$ are denoted by $\mathbf{h}_k$. A valid code word is denoted by $\mathbf{x} = (x_0, x_1, ..., x_{N-1})$ and the received log-likelihood ratios (LLRs) by $\mathbf{y} = (y_0, y_1, ..., y_{N-1})$

A trellis diagram is a graphical representation of the code word space of a channel code. Every path in a trellis connecting the starting and end point correspond to exactly one code word.

The trellis diagram for block codes can be constructed in two different ways: using the generator matrix or the parity check matrix (PCM) of the code.

The first construction method is based on the generator matrix (see Lin and Jr (2004) for more information). Since this construction method is quite complex and requires the generator matrix to be in a special form (trellis oriented generator matrix form), we apply the second construction method based on the PCM.

The PCM method provides full flexibility on the matrix, so that *any* PCM can be used. Furthermore, the trellis structure can easily be deduced from the PCM, which makes it suitable for a hardware implementation. Since the trellis describes a block code, we call it a bit-level trellis – in contrast to the convolutional trellis. In the following, we describe the construction method of Wolf (1978).

The bit-level trellis has N trellis steps (sometimes called time steps), and $M = 2^{N-K}$ states per step. The states are labelled by a binary $N-K$ tuple $\mathbf{s}_k^m$, where $m = 0, 1, ..., M-1$ denotes the index of a state in step $k = 0, 1, ..., N$.

To construct the trellis, the states between step k and $k+1$ are connected by branches. There are two different types of branches. One corresponds to code bit '0' ($x_k = 0$) the other one to code bit '1' ($x_k = 1$). The branches are established recursively as follows: From each state $\mathbf{s}_k^m$ of step k two branches depart to state

$$\mathbf{s}_{k+1}^m = \mathbf{s}_k^m \qquad for\ \ x_k = 0$$

$$\mathbf{s}_{k+1}^l = \mathbf{s}_k^m + \mathbf{h}_k \quad for\ \ x_k = 1$$

Since only paths from $\mathbf{s}_0^0$ to $\mathbf{s}_N^0$ represent code words, all other paths can be expurgated.

There are some remarkable differences between a convolutional trellis and a bit-level trellis. A bit-level trellis is in general a time-varying trellis, i.e. state transitions change over time – in contrast to the convolutional trellis.

Besides the time varying property, in a bit-level trellis only one bit is associated with a time step. In a convolutional trellis usually two or more bits correspond to one time step.

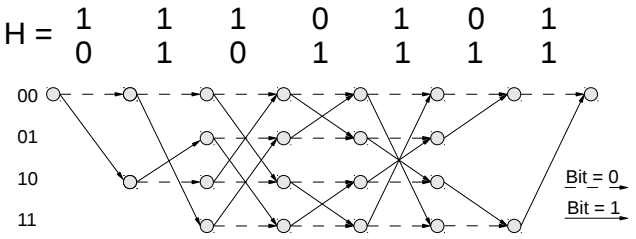

Fig. 1. Example for a trellis construction using the PCM $\mathbf{H}$

Additionally, it should be pointed out that the '0' branches always connect states having the same label. This property can be exploited in hardware, as we will see later.

A small example for $N = 7$ and $K = 5$ with a $M = 4$ state trellis is depicted in Fig. 1 to clarify the trellis construction.

3 Trellis Decoding Algorithms

Once the trellis representation of a code is obtained, it enables the use of very efficient decoding algorithms, like the VA and the BCJR algorithm. In this section we give a short summary of these two algorithms for bit-level trellises and their variants in the log domain.

3.1 Viterbi Algorithm

The VA (Viterbi (1967)) performs ML decoding efficiently on a trellis. It looks for the most probable path in the trellis by recursively building up paths though the trellis and discarding unlikely paths in every step. We shortly repeat the VA in the logarithm domain, that is usually used for hardware implementation.

For the VA every trellis state $\mathbf{s}_k^m$ is assigned a state metric $\alpha(\mathbf{s}_k^m)$. The state metrics for step $k+1$ are calculated recursively from those of step k.

1. initialize the state metrics at step 0: $\mathbf{s}_0^0$ with 0 and all other $\mathbf{s}_0^m$ ($m \neq 0$) with -infinity
2. For all states $k = 0, 1, .., N-1$ and all indices $m = 0, 1.., M-1$ calculate

$$\alpha\left(\mathbf{s}_k^l\right) = max\left[\alpha\left(\mathbf{s}_k^l\right),\ \alpha\left(\mathbf{s}_k^m\right) + y_k\right] \tag{1}$$

   where $\mathbf{s}_k^l = \mathbf{s}_k^m + \mathbf{h}_k$
3. output the path which lead to the maximum state metric $\alpha\left(\mathbf{s}_N^0\right)$, called the traceback step.

More detailed information on the VA can easily be found in literature (Lin and Jr (2004); Wolf (1978)).

3.2 BCJR and Max-log-MAP Algorithm

A drawback of the VA is that it does not provide any soft output information, which is required by modern decoding

heuristics. However, the BCJR algorithm (Bahl et al. (1974)) provides such additional information.

For hardware implementations is it advantageous to use the BCJR algorithm in the logarithm domain, which is called log-MAP, or its low complexity version max-log-MAP (Robertson et al. (1995)). In this paper we consider the max-log-MAP because it provides low complexity without degrading the decoding performance significantly.

In the max-log-MAP (and also the BCJR algorithm) every state is assigned two state metrics: the forward state metrics $\alpha(\mathbf{s}_k^m)$ and the backward state metrics $\beta(\mathbf{s}_k^m)$.

Max-log-MAP decoding consists of three phases:

1. forward recursion (calculates state metrics $\alpha(\mathbf{s}_k^m)$)
2. backward recursion (calculates state metrics ($\beta(\mathbf{s}_k^m)$)
3. soft output calculation (using $\alpha(\mathbf{s}_k^m)$, $\beta\left(\mathbf{s}_{k+1}^m\right)$ and y_k)

The forward recursion is equal to that of the VA in Eq. 1. For the backward recursion the states are processed in reversed order, i.e. from right to left. Details of the algorithm can be found e.g. in Lin and Jr (2004).

It can already be seen, that the recursion steps are important in both the VA and the max-log-MAP. It consists of add-compare-select (ACS) operations in Eq. 1 and requires permutations of state metrics according to the time varying branch structure. In the following section, we will focus on the hardware architecture for such a recursion unit for bit-level trellises.

4 Proposed Architectures

In this section we present the hardware architectures for VA and the max-log-MAP decoder for bit-level trellises. We propose the use of a Banyan network, that to implement the flexible state transitions. Finally we present a folded architecture, which reduces the decoder area to enable the implementation for trellises with a large number of states.

4.1 Architectures for Viterbi and Max-Log-MAP algorithm

The top level architectures for Viterbi and max-log-MAP decoder are depicted in Fig. 2 and 3. Although they look different, their core functionality is the recursion unit, which in both cases is identical.

The recursion unit for a simple case of a 4 state trellis is shown in Fig. 4. All state metrics $\alpha(\mathbf{s}_k^m)$ of one time step k are calculated in parallel. The different trellis steps are calculated consecutively, i.e. one trellis step per clock cycle. The state metrics are temporarily stored in the registers.

One iteration consists of routing the state metrics through the permutation network first. The routing is done according to the branches to the ACS units and is determined by the

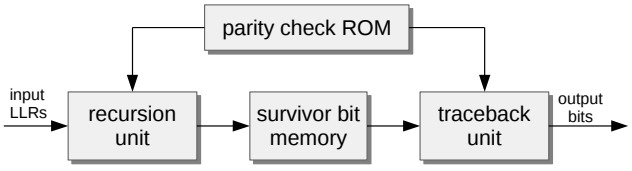

Fig. 2. Top level architecture of the Viterbi decoder

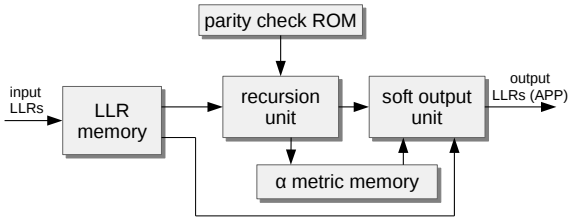

Fig. 3. Top level architecture for the max-log-MAP decoder

columns of the PCM $\mathbf{h}_k$. The ACS calculates Eq. 1. Finally the state metrics are stored in the register again.

Note, that the state metrics are directly fed back to the register. This maps the branches for code bit '0' to the hardware.

The state metrics and received LLRs $\mathbf{y}$ are represented as fixed point numbers. To keep the number of bits low and therefore to save resources without provoking catastrophic overflows, modulo arithmetic is used here (Hekstra (1989)).

4.2 Permutation Network

An essential part of the trellis decoders is the permutation network (PN). It routes the state metrics along the Bit '1' branches during the recursion.

The PNs in the trellis decoders must have $M = 2^{N-K}$ inputs and outputs. The network is controlled by the column

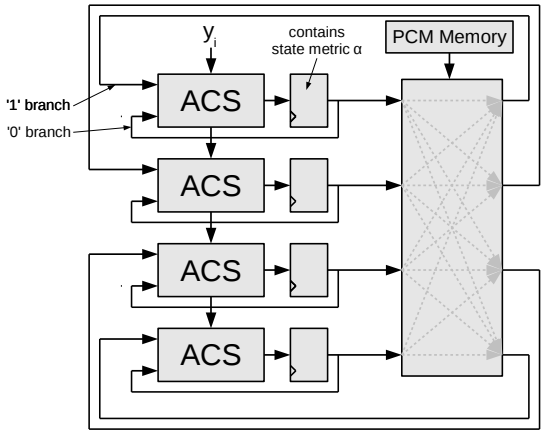

Fig. 4. Architecture of the recursion unit

bits of the PCM $\mathbf{h}_k$ and consists of 2×2 butterfly switches, which can permute or route through the two inputs.

We first investigate the use of the well known Benes PN (Benes (1964)) and in a second step propose the use of a more sophisticated network, the Banyan PN (Goke and Lipovski (1973)).

The Benes PN is capable of performing all possible $M!$ permutations, although in this application only M are required. Therefore, the Benes PN uses more resources than necessary. A drawback of the Benes PN is its elevated number of $2 \cdot log_2(M) - 1$ stages, which directly lengthens the maximum combinational path of the recursion unit and thus decreases the throughput. Furthermore the Benes PN requires a complex controlling logic, which maps the bits from the PCM to the control bits of the switches, see Fig. 7.

The second network studied is the Banyan PN. It performs the necessary permutation with approximately half the number of stages ($log_2(M)$) in comparison to the Benes PN. This reduces the signal propagation time and improves throughput. Moreover the Banyan PN has approximately half the number of switches (see Fig. 5). Another advantage is that the bits of the columns of the PCM can directly be applied to control the switches in the network. No extra controlling logic is required. The small number of switches and the saving of the complex controlling logic makes the Banyan PN the network of choice for this application.

In the following, we provide our method of how to build up such a network. The Banyan PN is constructed from two Butterfly PNs and a final permutation stage. The smallest Butterfly PN is composed of a single switch and larger ones can be constructed recursively, i.e. a network with M inputs is obtained from two Butterfly PNs with $M/2$ inputs.

We place the second Butterfly PN below the first one and denote the outputs by $S(i)$ with $i = 0, 1, .., M/2-1$ and $i = M/2, .., M-1$. The final stage is a column of $M/2$ switches that placed right of the two Butterfly PNs. Their inputs are denoted by $I(i)$ with $i = 0, 1, .., M-1$ and their outputs by $B(i)$. The connections between the two Butterfly PNs and the final stage is done by the following algorithm:

Algorithm 1 Intermediate connections of a Butterfly PN

for $(i = 0$ **to** $(M/4-1))$ **do**
  $I(2*i) \leftarrow S(2*i)$
  $I(2*i+1) \leftarrow S(2*i+M/2)$
  $I(2*i+M/2) \leftarrow S(2*i+1)$
  $I(2*i+M/2+1) \leftarrow S(2*i+M/2+1)$
end for

Finally, we connect the outputs $B(i)$ of the final stage to the outputs $O(i)$ of the Banyan PN by using the following algorithm:

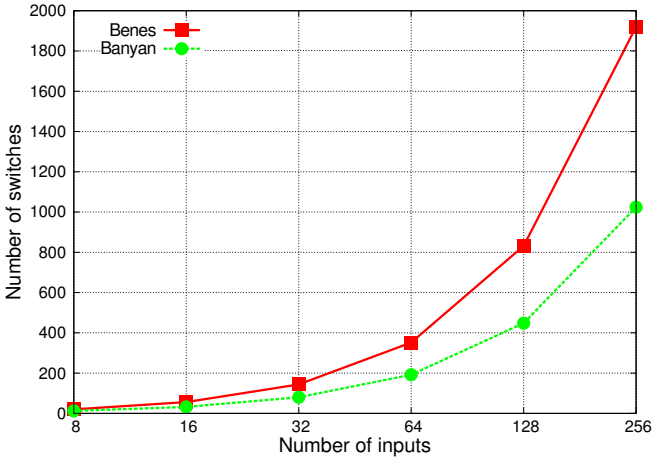

Fig. 5. Number of switches used by the Benes and the Banyan PN

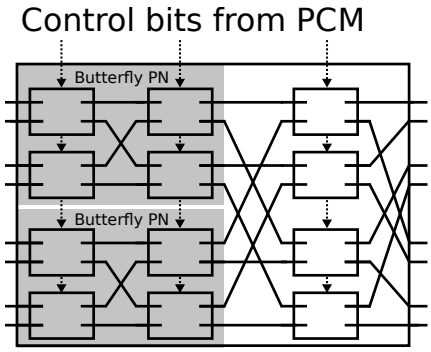

Fig. 6. Eight input Banyan PN: it needs less switches than the Benes PN and no controlling is needed

Algorithm 2 Output connections of a Banyan PN

for $(i = 0$ **to** $(n/2-1))$ **do**
  $O(i) \leftarrow B(2*i)$
  $O(i+n/2) \leftarrow B(2*i+1)$
end for

Each input of the network has to be able to reach all the M different output addresses and that is only possible with $N-K$ stages. Thus it is impossible to do all the required permutations with a network with less stages than the Banyan.

Fig. 6 shows the construction of an 8x8 Banyan PN using the algorithms from above.

4.3 Folding

For trellises with a large number of states M, the above presented architecture grows quickly and may become too large for an FPGA implementation. To counteract this problem, we propose folding to reduce the required resources of the decoder at the expense of a reduced throughput.

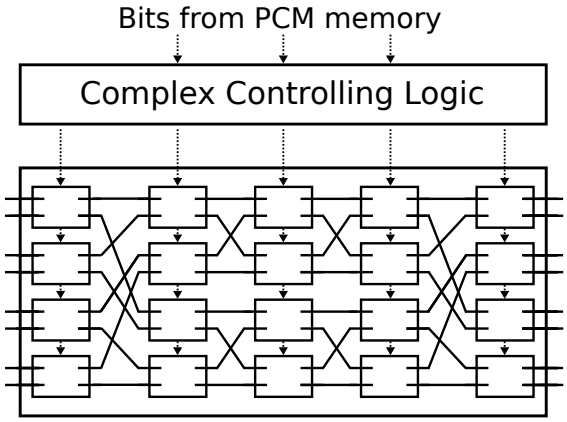

Fig. 7. Eight input Benes PN: it needs more switches than the Banyan PN and additional controlling is needed

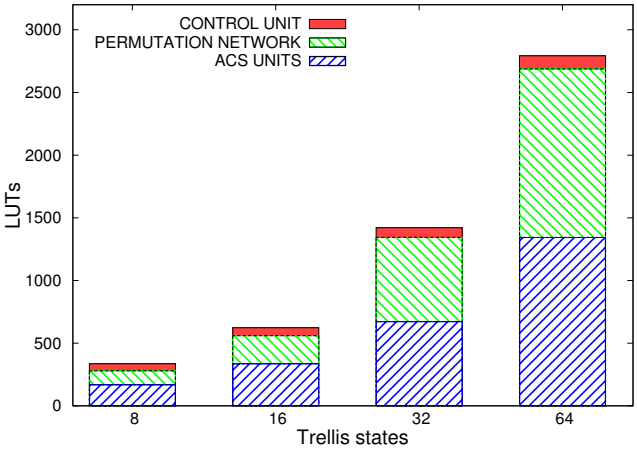

Fig. 8. Viterbi decoder: look-up tables

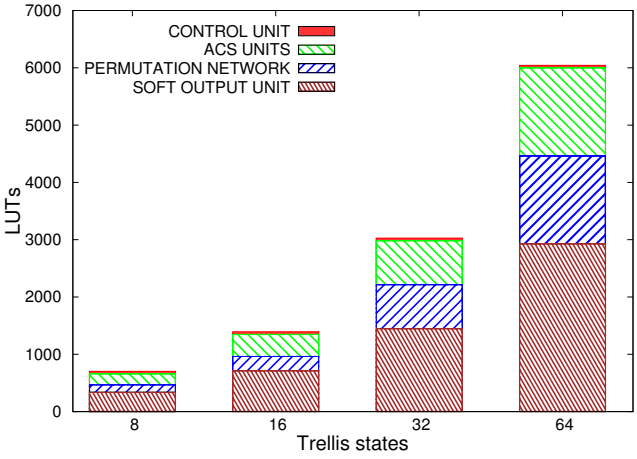

Fig. 9. Max-log-MAP decoder : look-up tables

In the folded architecture not all M state metrics in one trellis step are calculated in one clock cycle. Instead the state metrics are calculated, e.g. in two clock cycles. In this case half of the ACS units can be reused.

Furthermore it reduces the size of the PN by more than half given that its construction is recursive (see above). Note, that the folded Banyan PN with $M/2$ exactly fulfils the permutation requirements of the folded architecture, and is thus suitable for folded architectures.

The number of clock cycles required to calculate the state metrics of one trellis step is called folding factor f and must be a power of 2.

Due to folding the size of the PN, the number of ACS units and the soft output unit (for the max-log-MAP decoder) is largely reduced. However, some additional hardware resources are needed to distribute the state metrics to the PN and ACS units and registers to store the partially calculated state metrics.

The area reduction of the soft output unit is directly proportional to the folding factor f. Therefore folding is especially advantageous for the max-log-MAP decoder.

5 FPGA Implementation

To evaluate the architectures in detail we have implemented the architectures on a Xilinx Virtex 6 (XC6VLX75T-3) FPGA using ISE 14.1. The architectures for the VA, the max-log-MAP and their folded version have been evaluated for trellises with different number of states. We analyse the resource consumption regarding the required look-up tables (LUTs) after place & route.

The numbers presented are only dependent on the number of trellis states M and not on the number of trellis steps N. The number of trellis steps N only influences the size of the memories. However this is not the critical resource in the design.

The quantization of the metrics is dependent on the context in which the decoders are used. However it has been experienced by simulations that a state metric quantization of 7 Bits for the VA and 8 Bits for the max-log-MAP is reasonable.

In Fig. 8 and 9 the required LUTs for the unfolded architecture ($f = 1$) is shown, separated for each decoder part.

For the VA almost 50% of the LUTs are occupied by the ACS units. The PN also consumes nearly 50% of the LUTs. Therefore the ACS units and the PN are the dominating parts. The remaining fraction is occupied by the controlling, which also includes the traceback unit.

The max-log-MAP is dominated by the soft output unit, which needs approximately 50% of all LUTs. The PN and the ACS units consume around 25% each.

In Fig. 10 and 11 the number of required LUTs for different folded architectures are shown. It shows how the occupied resources reduce and allows for the implementation of larger trellises. The throughput decrease for the VA can be seen in Fig. 12.

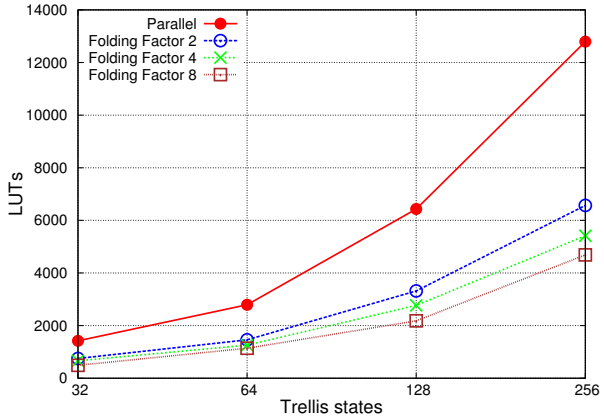

Fig. 10. Viterbi decoder: LUTs dependent on folding factor

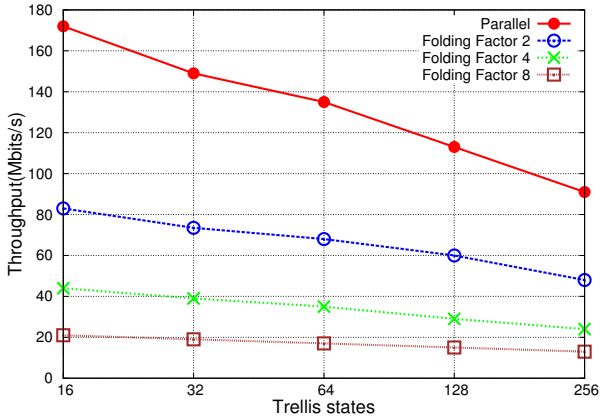

Fig. 12. Viterbi decoder: throughput dependent on folding factor

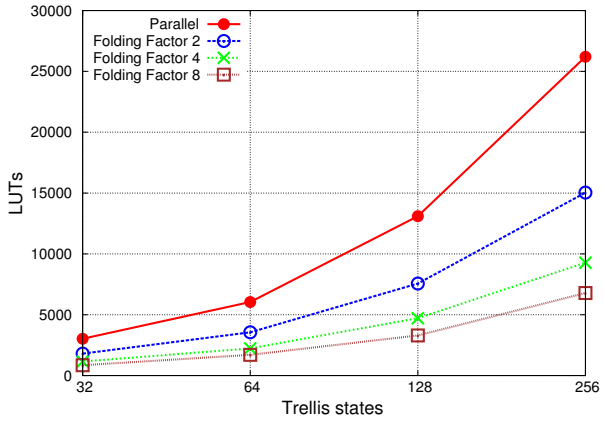

Fig. 11. Max-log-MAP decoder: LUTs dependent on folding factor

6 Conclusions

In this paper we have investigated hardware architectures for bit-level trellises. We have selected the trellis construction based on the PCM, because it provides maximum flexibility. After a brief review of the VA and the BCJR decoding algorithms, we proposed efficient hardware architectures. A key feature is the Banyan PN, which maps the time variant property of the trellis efficiently to hardware and significantly outperforms standard solutions like the Benes PN. Furthermore we presented a folded version of the architecture to enable the implementation for large trellises. Finally the resource consumption and throughput of the architecture have been evaluated on a Xilinx Virtex 6 FPGA.

Acknowledgements. We gratefully acknowledge partially financial support by the DFG (project-ID: KI 1754/1-1) as well as by the Center of Mathematical and Computational Modelling of the University of Kaiserslautern. We thank Frank Kienle for his valuable comments and suggestions.

References

Bahl, L., Cocke, J., Jelinek, F., and Raviv, J.: Optimal Decoding of Linear Codes for Minimizing Symbol Error Rate, IEEE Transaction on Information Theory, IT-20, 284–287, 1974.

Benes, V. E.: Optimal Rearrangeable Multistage Connecting Networks, The Bell System Technical Journal, 4, 1964.

Goke, L. R. and Lipovski, G. J.: Banyan Networks for Partitioning Multiprocessor Systems, 1st Annual Symposium on Computer Architecture, pp. 21–28, 1973.

Hekstra, A. P.: An Alternative to Metric Rescaling in Viterbi Decoders, IEEE Transactions on Communications, 37, 1220–1222, doi:http://dx.doi.org/10.1109/26.4651610.1109/26.46516, 1989.

Kim, S., Ryoo, S., and Lee, S.: Block Turbo Codes Using Multiple Soft Outputs, in: Proceedings of the 3rd ISTC, vol. 1, pp. 247–250, Brest, 2003.

Lin, S. and Jr, D. C.: Error Control Coding 2nd., Prentice Hall PTR, Upper Saddle River, New Jersey, USA, 2004.

Liu, C. Y. and Lin, S.: Turbo encoding and decoding of Reed-Solomon codes through binary decomposition and self-concatenation, IEEE Transactions on Communications, 52, 1484–1493, 2004.

Pyndiah, R., Glavieux, A., Picart, A., and Jacq, S.: Near optimum decoding of product codes, in: Proc. IEEE Global Telecommunications Conf. GLOBECOM '94. Communications: The Global Bridge, pp. 339–343, 1994.

Robertson, P., Villebrun, E., and Hoeher, P.: A Comparison of Optimal and Sub-Optimal MAP decoding Algorithms Operating in the Log-Domain, in: Proc. 1995 International Conference on Communications (ICC '95), pp. 1009–1013, Seattle, Washington, USA, 1995.

Tanner, R. M.: A Recursive Approach to Low Complexity Codes, IEEE Transaction on Information Theory, IT-27, 533–547, 1981.

Vardy, A. and Be'ery, Y.: Bit-level soft-decision decoding of Reed-Solomon codes, IEEE Transactions on Communications, 39, 440–444, 1991.

Viterbi, A. J.: Error Bounds for Convolutional Codes and an Asymptotically Optimum Decoding Algorithm, IEEE Transactions on Information Theory, 13, 260–269, 1967.

Wolf, J.: Efficient maximum likelihood decoding of linear block codes using a trellis, IEEE Transactions on Information Theory, 24, 76–80, 1978.

APPENDIX C VHDL CODE

C.1 Block Codes Trellis Decoders Package

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

package trellis_decoders_bc_lib is

constant SM_BITS    : integer := 7;
constant LLR_BITS    : integer := 5;
constant N : integer := 255; --CODE LENGHT
constant ADDRESS_BITS : integer := 8; -- log2(N) -> rounded up
constant R : integer := 8; --REDUNDANT BITS = N - K
constant NUMBER_OF_SM : integer := 2**R;

type state_metrics_array_type is array (0 to NUMBER_OF_SM -1)
of std_logic_vector (SM_BITS -1 downto 0);
type state_metric_bits_type is array (0 to SM_BITS -1)
of std_logic_vector(0 to NUMBER_OF_SM -1);
type permutation_network_internal_connections is array(0 to R-1)
of std_logic_vector(0 to NUMBER_OF_SM);

type PN_2x2_IO_type is array (0 to 1)
of std_logic_vector (SM_BITS -1 downto 0);
type PN_4x4_IO_type is array (0 to 3)
of std_logic_vector (SM_BITS -1 downto 0);
type PN_8x8_IO_type is array (0 to 7)
of std_logic_vector (SM_BITS -1 downto 0);
type PN_16x16_IO_type is array (0 to 15)
of std_logic_vector (SM_BITS -1 downto 0);
type PN_32x32_IO_type is array (0 to 31)
of std_logic_vector (SM_BITS -1 downto 0);
type PN_64x64_IO_type is array (0 to 63)
of std_logic_vector (SM_BITS -1 downto 0);
type PN_128x128_IO_type is array (0 to 127)
of std_logic_vector (SM_BITS -1 downto 0);
type PN_256x256_IO_type is array (0 to 255)
of std_logic_vector (SM_BITS -1 downto 0);
type PN_512x512_IO_type is array (0 to 511)
of std_logic_vector (SM_BITS -1 downto 0);
type PN_1024x1024_IO_type is array (0 to 1023)
of std_logic_vector (SM_BITS -1 downto 0);

component Recursion_Unit
  port (
```

```
    CLOCK : in std_logic;
    RECURSION_H_COLUMN : in std_logic_vector (R-1 downto 0);
    LLR : in std_logic_vector (LLR_BITS-1 downto 0);
    RESET_SM : in std_logic;
    WRITE_SM_REGISTERS : in std_logic;
    CHOOSEN_BITS : out std_logic_vector (NUMBER_OF_SM -1 downto 0)
  );
end component;

component Survivor_Management_Unit
    port(
  CLOCK : in std_logic;
  CHOOSEN_BITS : in std_logic_vector (NUMBER_OF_SM -1 downto 0);
  TRACEBACK_H_COLUMN : in std_logic_vector (R-1 downto 0);
  ENABLE_TRACEBACK_MEMORY : in std_logic;
  TRACEBACK_MEMORY_A_ADDRESS
  : in std_logic_vector(ADDRESS_BITS -1 downto 0);
  TRACEBACK_MEMORY_B_ADDRESS
  : in std_logic_vector(ADDRESS_BITS -1 downto 0);
  TRACEBACK_MEMORY_A_WRITE_ENABLE : in  std_logic;
  TRACEBACK_MEMORY_B_WRITE_ENABLE : in std_logic;
  RESET_TRACEBACKING_MEMORY_REGISTER : in std_logic;
  TOGGLE_TRACEBACKING_MEMORY: in std_logic;
  RESET_METRIC_INDEX_REGISTER: in std_logic;
  TRACEBACKING_MEMORY : out std_logic;
  DECODED_BIT : out std_logic
    );
end component;

component LLR_Unit is
  port (
    CLOCK : in std_logic;
    ALPHA_VALUES : in state_metrics_array_type;
    BETA_VALUES_1_TRANSITIONS : in state_metrics_array_type;
    BETA_VALUES_0_TRANSITIONS : in state_metrics_array_type;
    LLR :  in std_logic_vector (LLR_BITS -1 downto 0);
    WRITE_PIPELINE_REGS : in std_logic;
    RESET_MAP_REGISTER : in std_logic;
    WRITE_RESULT : in std_logic;
    MAP_RESULT : out  STD_LOGIC_VECTOR (SM_BITS -1 downto 0)
  );
end component;

component ACS
    Port ( STATE_METRIC_X : in  STD_LOGIC_VECTOR (SM_BITS -1  downto 0);
           STATE_METRIC_Y : in  STD_LOGIC_VECTOR (SM_BITS -1 downto 0);
           LLR : in  STD_LOGIC_VECTOR (LLR_BITS-1 downto 0);
           CHOOSEN_BIT : out  STD_LOGIC;
           SELECTED_METRIC : out  STD_LOGIC_VECTOR (SM_BITS -1 downto 0));
end component;

component H_Matrix_ROM
    Port ( CLOCK : in  STD_LOGIC;
           ENABLE : in  STD_LOGIC;
           ADDRESS_A : in  STD_LOGIC_VECTOR (ADDRESS_BITS -1 downto 0);
     ADDRESS_B : in  STD_LOGIC_VECTOR (ADDRESS_BITS -1 downto 0);
           DATA_OUT_A : out  STD_LOGIC_VECTOR (R-1 downto 0);
     DATA_OUT_B : out  STD_LOGIC_VECTOR (R-1 downto 0));
```

```
end component;

component Traceback_Memory
    Port ( CLOCK : in  STD_LOGIC;
           ENABLE : in  STD_LOGIC;
           WRITE_ENABLE : in  STD_LOGIC;
           DATA_IN : in  STD_LOGIC_VECTOR (NUMBER_OF_SM -1 downto 0);
           ADDRESS : in  STD_LOGIC_VECTOR (ADDRESS_BITS -1 downto 0);
           DATA_OUT : out  STD_LOGIC_VECTOR (NUMBER_OF_SM -1 downto 0)
   );
end component;

component LLR_Memory
    Port ( CLOCK : in  STD_LOGIC;
           ENABLE : in  STD_LOGIC;
     WRITE_ENABLE : in  STD_LOGIC;
           ADDRESS : in  STD_LOGIC_VECTOR (ADDRESS_BITS -1 downto 0);
     DATA_IN : in  STD_LOGIC_VECTOR (LLR_BITS -1 downto 0);
           DATA_OUT : out  STD_LOGIC_VECTOR (LLR_BITS -1 downto 0));
end component;

component Alpha_Values_Memory
    Port ( CLOCK : in  STD_LOGIC;
     ENABLE : in  STD_LOGIC;
     WRITE_ENABLE : in  STD_LOGIC;
           ADDRESS : in  STD_LOGIC_VECTOR (ADDRESS_BITS -1 downto 0);
     DATA_IN : in
     STD_LOGIC_VECTOR ((NUMBER_OF_SM)*(SM_BITS) -1 downto 0);
           DATA_OUT : out
     STD_LOGIC_VECTOR ((NUMBER_OF_SM)*(SM_BITS) -1 downto 0));
end component;

component Three_Elements_Adder -- adds the elements of a one-transition
    Port ( ALFA_METRIC : in  STD_LOGIC_VECTOR (SM_BITS -1 downto 0);
           BETA_METRIC : in  STD_LOGIC_VECTOR (SM_BITS -1 downto 0);
           LLR : in  STD_LOGIC_VECTOR (LLR_BITS -1 downto 0);
           RESULT : out  STD_LOGIC_VECTOR (SM_BITS -1 downto 0));
  end component;

component Two_Elements_Adder is
    Port ( ALFA_METRIC : in  STD_LOGIC_VECTOR (SM_BITS -1 downto 0);
           BETA_METRIC : in  STD_LOGIC_VECTOR (SM_BITS -1 downto 0);
           RESULT : out  STD_LOGIC_VECTOR (SM_BITS -1 downto 0));
end component;

component Lower_Value_Selector
    Port ( CLOCK : in std_logic;
     INPUT_VALUES : in  state_metrics_array_type;
     WRITE_PIPELINE_REGS : in std_logic;
           LOWER_VALUE : out  STD_LOGIC_VECTOR (SM_BITS -1 downto 0));
end component;


component Banyan_Permutation_Network is
    Port ( INPUTS : in  state_metrics_array_type;
     CONTROLS : in std_logic_vector(0 to R -1);
           OUTPUTS : out  state_metrics_array_type);
end component;
```

```
component SM_Switch
    Port (  SM_A : in  STD_LOGIC_VECTOR (SM_BITS -1 downto 0);
            SM_B : in  STD_LOGIC_VECTOR (SM_BITS -1 downto 0);
            CONTROL : in  STD_LOGIC;
            OUT_A : out  STD_LOGIC_VECTOR (SM_BITS -1 downto 0);
            OUT_B : out  STD_LOGIC_VECTOR (SM_BITS -1 downto 0));
end component;

component Butterfly_Network_4x4 is
    Port ( INPUTS : in  PN_4x4_IO_type;
           CONTROLS : in  STD_LOGIC_VECTOR (0 to 1);
           OUTPUTS : out  PN_4x4_IO_type
   );
end component;

component Banyan_Network_4x4 is
    Port ( INPUTS : in  PN_4x4_IO_type;
           CONTROLS : in  STD_LOGIC_VECTOR (0 to 1);
           OUTPUTS : out  PN_4x4_IO_type
   );
end component;

component  Butterfly_Network_8x8 is
 Port ( INPUTS : in  PN_8x8_IO_type;
           CONTROLS : in  STD_LOGIC_VECTOR (0 to 2);
           OUTPUTS : out  PN_8x8_IO_type);
end component;

component  Banyan_Network_8x8 is
 Port ( INPUTS : in  PN_8x8_IO_type;
           CONTROLS : in  STD_LOGIC_VECTOR (0 to 2);
           OUTPUTS : out  PN_8x8_IO_type);
end component;

component  Butterfly_Network_16x16 is
 Port ( INPUTS : in  PN_16x16_IO_type;
           CONTROLS : in  STD_LOGIC_VECTOR (0 to 3);
           OUTPUTS : out  PN_16x16_IO_type);
end component;

component  Banyan_Network_16x16 is
 Port ( INPUTS : in  PN_16x16_IO_type;
           CONTROLS : in  STD_LOGIC_VECTOR (0 to 3);
           OUTPUTS : out  PN_16x16_IO_type);
end component;

component  Butterfly_Network_32x32 is
 Port ( INPUTS : in  PN_32x32_IO_type;
           CONTROLS : in  STD_LOGIC_VECTOR (0 to 4);
           OUTPUTS : out  PN_32x32_IO_type);
end component;

component  Banyan_Network_32x32 is
 Port ( INPUTS : in  PN_32x32_IO_type;
           CONTROLS : in  STD_LOGIC_VECTOR (0 to 4);
           OUTPUTS : out  PN_32x32_IO_type);
end component;
```

```
component  Butterfly_Network_64x64 is
 Port ( INPUTS : in  PN_64x64_IO_type;
           CONTROLS : in  STD_LOGIC_VECTOR (0 to 5);
           OUTPUTS : out  PN_64x64_IO_type);
end component;

component  Banyan_Network_64x64 is
 Port ( INPUTS : in  PN_64x64_IO_type;
           CONTROLS : in  STD_LOGIC_VECTOR (0 to 5);
           OUTPUTS : out  PN_64x64_IO_type);
end component;

component  Butterfly_Network_128x128 is
 Port ( INPUTS : in  PN_128x128_IO_type;
           CONTROLS : in  STD_LOGIC_VECTOR (0 to 6);
           OUTPUTS : out  PN_128x128_IO_type);
end component;

component  Banyan_Network_128x128 is
 Port ( INPUTS : in  PN_128x128_IO_type;
           CONTROLS : in  STD_LOGIC_VECTOR (0 to 6);
           OUTPUTS : out  PN_128x128_IO_type);
end component;

component  Butterfly_Network_256x256 is
 Port ( INPUTS : in  PN_256x256_IO_type;
           CONTROLS : in  STD_LOGIC_VECTOR (0 to 7);
           OUTPUTS : out  PN_256x256_IO_type);
end component;

component  Banyan_Network_256x256 is
 Port ( INPUTS : in  PN_256x256_IO_type;
           CONTROLS : in  STD_LOGIC_VECTOR (0 to 7);
           OUTPUTS : out  PN_256x256_IO_type);
end component;

component  Butterfly_Network_512x512 is
 Port ( INPUTS : in  PN_512x512_IO_type;
           CONTROLS : in  STD_LOGIC_VECTOR (0 to 8);
           OUTPUTS : out  PN_512x512_IO_type);
end component;

component  Banyan_Network_512x512 is
 Port ( INPUTS : in  PN_512x512_IO_type;
           CONTROLS : in  STD_LOGIC_VECTOR (0 to 8);
           OUTPUTS : out  PN_512x512_IO_type);
end component;

component  Banyan_Network_1024x1024 is
 Port ( INPUTS : in  PN_1024x1024_IO_type;
           CONTROLS : in  STD_LOGIC_VECTOR (0 to 9);
           OUTPUTS : out  PN_1024x1024_IO_type);
end component;
end trellis_decoders_bc_lib;
```

C.2 Viterbi Decoder Top Level

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use work.trellis_decoders_bc_lib.ALL;

entity Viterbi_Decoder_Block_Codes is
   Port ( CLOCK : in  STD_LOGIC;
  RESET : in STD_LOGIC;
  LLR : in std_logic_vector(LLR_BITS -1 downto 0);
  DECODED_BIT : out std_logic
   );
end Viterbi_Decoder_Block_Codes;

architecture Behavioral of Viterbi_Decoder_Block_Codes is

--WIRES
signal choosen_bits : std_logic_vector (NUMBER_OF_SM -1 downto 0);
signal recursion_H_column : std_logic_vector (R-1 downto 0);
signal traceback_H_column : std_logic_vector (R-1 downto 0);
signal decoded_bit_internal_signal :  std_logic;
signal tracebacking_memory :  std_logic;
--REGISTERS
signal up_counter : std_logic_vector(ADDRESS_BITS -1 downto 0);
signal down_counter : std_logic_vector(ADDRESS_BITS -1 downto 0);
-- two addresses to acess the dual port H memory ROM
signal H_memory_address_A_register : std_logic_vector(ADDRESS_BITS -1 downto 0);
signal H_memory_address_B_register : std_logic_vector(ADDRESS_BITS -1 downto 0);
--CONTROL SIGNALS
signal reset_sm : std_logic;
signal write_sm_registers : std_logic;
signal enable_H_memory : std_logic;
signal enable_traceback_memory : std_logic;
signal traceback_memory_A_address : std_logic_vector(ADDRESS_BITS -1 downto 0);
signal traceback_memory_B_address : std_logic_vector(ADDRESS_BITS -1 downto 0);
signal traceback_memory_A_write_enable : std_logic;
signal traceback_memory_B_write_enable : std_logic;
signal reset_tracebacking_memory_register : std_logic;
signal toggle_tracebacking_memory: std_logic;
signal reset_metric_index_register: std_logic;
signal reset_up_counter : std_logic;
signal reset_down_counter : std_logic;
signal increment_H_memory_address_A  : std_logic;
signal decrement_H_memory_address_B : std_logic;
signal reset_memory_address : std_logic;
signal count_up : std_logic;
signal count_down : std_logic;

-- FSM
type state_type is
(RESET_STATE, FWD_RECURSION, PREPARE_TRACEBACK, TRACEBACK_AND_FWD_RECURSION);
signal state, next_state : state_type;

begin
```

```
H_memory: H_Matrix_ROM port map(
  CLOCK => CLOCK,
  ENABLE => enable_H_memory,
  ADDRESS_A => H_memory_address_A_register,
  ADDRESS_B => H_memory_address_B_register,
  DATA_OUT_A => recursion_H_column,
  DATA_OUT_B => traceback_H_column
);

-- address used in forward recursion
H_memory_address_A_reg: process (CLOCK, reset_memory_address)
begin
  if (reset_memory_address = '1') then
    H_memory_address_A_register <= (others => '0');
  elsif falling_edge (CLOCK) then
    if (increment_H_memory_address_A = '1') then
      H_memory_address_A_register <= H_memory_address_A_register + '1';
    else
      H_memory_address_A_register <= H_memory_address_A_register;
    end if;
  end if;
end process;

-- address used in traceback
H_memory_address_B_reg: process (CLOCK, reset_memory_address)
begin
  if (reset_memory_address = '1') then
    H_memory_address_B_register <= conv_std_logic_vector(N-1, ADDRESS_BITS);
  elsif falling_edge (CLOCK) then
    if (decrement_H_memory_address_B = '1') then
      H_memory_address_B_register <= H_memory_address_B_register - '1';
    else
      H_memory_address_B_register <= H_memory_address_B_register;
    end if;
  end if;
end process;

RU: Recursion_Unit port map(
  CLOCK => CLOCK,
  RECURSION_H_COLUMN => recursion_H_column,
  LLR => LLR,
  RESET_SM => reset_sm,
  WRITE_SM_REGISTERS => write_sm_registers,
  CHOOSEN_BITS => choosen_bits
);

SMU: Survivor_Management_Unit port map(
  CLOCK => CLOCK,
  CHOOSEN_BITS => choosen_bits,
  TRACEBACK_H_COLUMN => traceback_H_column,
  ENABLE_TRACEBACK_MEMORY => enable_traceback_memory,
  TRACEBACK_MEMORY_A_ADDRESS => traceback_memory_A_address,
  TRACEBACK_MEMORY_B_ADDRESS => traceback_memory_B_address,
  TRACEBACK_MEMORY_A_WRITE_ENABLE => traceback_memory_A_write_enable,
  TRACEBACK_MEMORY_B_WRITE_ENABLE => traceback_memory_B_write_enable,
  RESET_TRACEBACKING_MEMORY_REGISTER => reset_tracebacking_memory_register,
  TOGGLE_TRACEBACKING_MEMORY => toggle_tracebacking_memory,
  RESET_METRIC_INDEX_REGISTER => reset_metric_index_register,
```

```
  TRACEBACKING_MEMORY => tracebacking_memory,
  DECODED_BIT => decoded_bit_internal_signal
);

DECODED_BIT <= decoded_bit_internal_signal;
------------------------------- FSM ----------------------------

process (CLOCK, RESET, state)
begin
  if (RESET = '1') then
    state <= RESET_STATE;
  elsif rising_edge(CLOCK) then
    state <= next_state;
  else
    state <= state;
  end if;
end process;

process (state, up_counter, down_counter)
begin
  case state is
    when RESET_STATE =>
      next_state <= FWD_RECURSION;
    when FWD_RECURSION =>
      if (up_counter  = conv_std_logic_vector(N-2, ADDRESS_BITS)) then
        next_state <= PREPARE_TRACEBACK;
      else
        next_state <= FWD_RECURSION;
      end if;
    when PREPARE_TRACEBACK =>
      next_state <= TRACEBACK_AND_FWD_RECURSION;
    when TRACEBACK_AND_FWD_RECURSION =>
      if (up_counter  = conv_std_logic_vector(N-2, ADDRESS_BITS))then
        next_state <= PREPARE_TRACEBACK;
      else
        next_state <= TRACEBACK_AND_FWD_RECURSION;
      end if;
    when others =>
      next_state <= RESET_STATE;
  end case;
end process;

-- auxiliary counters
up_counter_process: process (CLOCK)
begin
  if rising_edge (CLOCK) then
    if (reset_up_counter = '1') then
      up_counter <= (others => '0');
    elsif (count_up = '1') then
      up_counter <= up_counter + '1';
    else
      up_counter <= up_counter;
    end if;
  end if;
end process;

down_counter_process: process (CLOCK)
begin
```

```
  if falling_edge (CLOCK) then
    if (reset_down_counter = '1') then
      down_counter <= conv_std_logic_vector(N-1, ADDRESS_BITS);
    elsif (count_down = '1') then
        down_counter <= down_counter - '1';
    else
      down_counter <= down_counter;
    end if;
  end if;
end process;

reset_sm <= '0' when
(state = FWD_RECURSION or state = TRACEBACK_AND_FWD_RECURSION) else '1';
write_sm_registers <= '1'
when (state = FWD_RECURSION or state = TRACEBACK_AND_FWD_RECURSION) else '0';
-- H ROM
enable_H_memory <=  '1';
increment_H_memory_address_A <= '1'
when (state = FWD_RECURSION or state = TRACEBACK_AND_FWD_RECURSION) else '0';
decrement_H_memory_address_B <= '1'
when (state = TRACEBACK_AND_FWD_RECURSION) else '0';
reset_memory_address <= '0'
when (state = FWD_RECURSION or state = TRACEBACK_AND_FWD_RECURSION) else '1';
-- TRACEBACK MEMORIES
enable_traceback_memory <= '0' when (state = RESET_STATE) else '1'; --'1'
when (state = FWD_RECURSION or state = TRACEBACK_AND_FWD_RECURSION) else '0';
traceback_memory_A_address <= down_counter when tracebacking_memory = '0'
else up_counter;
traceback_memory_B_address <= down_counter when tracebacking_memory = '1'
else up_counter;
traceback_memory_A_write_enable <= '1' when tracebacking_memory = '1' else '0';
traceback_memory_B_write_enable <= '1' when tracebacking_memory = '0' else '0';
reset_tracebacking_memory_register <= '1' when state = RESET_STATE else '0';
toggle_tracebacking_memory <= '1' when (state = PREPARE_TRACEBACK) else '0';
-- COUNTERS
reset_down_counter <= '0'
when (state = FWD_RECURSION or state = TRACEBACK_AND_FWD_RECURSION) else '1';
reset_up_counter <= reset_down_counter;
count_up <= '1'
when (state = FWD_RECURSION or state = TRACEBACK_AND_FWD_RECURSION) else '0';
count_down <= '1' when (state = TRACEBACK_AND_FWD_RECURSION) else '0';
reset_metric_index_register <= '0'
when state = TRACEBACK_AND_FWD_RECURSION else '1';
end Behavioral;
```

C.3 MAX-Log-MAP Top Level

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use work.trellis_decoders_bc_lib.ALL;

entity Max_Log_Map_Decoder_Block_Codes is
  Port ( CLOCK : in  STD_LOGIC;
         RESET : in  STD_LOGIC;
         LLR : in  STD_LOGIC_VECTOR (LLR_BITS -1 downto 0);
         MAP_RESULT : out  STD_LOGIC_VECTOR (SM_BITS -1 downto 0)
```

```
      );
end Max_Log_Map_Decoder_Block_Codes;

architecture Behavioral of Max_Log_Map_Decoder_Block_Codes is
--  WIRES
signal state_metrics : state_metrics_array_type;
signal pn_output : state_metrics_array_type;
signal alpha_values : state_metrics_array_type;
signal alpha_values_memory_output :
std_logic_vector ((NUMBER_OF_SM)*(SM_BITS) -1 downto 0);
signal alpha_values_memory_data_in :
std_logic_vector ((NUMBER_OF_SM)*(SM_BITS) -1 downto 0);
signal H_column : std_logic_vector (R-1 downto 0);
signal recursion_llr : std_logic_vector (LLR_BITS-1 downto 0);
signal llr_memory_output : STD_LOGIC_VECTOR (LLR_BITS-1 downto 0);
signal map_result_signal : STD_LOGIC_VECTOR (SM_BITS -1 downto 0);

--REGISTERS
signal up_down_counter : std_logic_vector(ADDRESS_BITS -1 downto 0);
signal up_counter : std_logic_vector(ADDRESS_BITS -1 downto 0);

--CONTROL SIGNALS
signal reset_sm : std_logic;
signal reset_map_register : std_logic;
signal write_sm_registers : std_logic;
signal recursion_llr_selector : std_logic;
signal enable_llr_memory : std_logic;
signal llr_memory_write_enable : std_logic;
signal llr_memory_address : std_logic_vector(ADDRESS_BITS -1 downto 0);
signal enable_H_memory : std_logic;
signal alpha_values_memory_address : std_logic_vector(ADDRESS_BITS -1 downto 0);
signal alpha_values_memory_enable : std_logic;
signal alpha_values_memory_write_enable : std_logic;
signal write_pipeline_regs : std_logic;
signal reset_counter : std_logic;
signal count_direction : std_logic;
signal reset_up_counter : std_logic;
signal count_up : std_logic;
signal write_result : std_logic;

--FSM
type state_type is (RESET_STATE, FWD_RECURSION, PREPARE_BACK_RECURSION,
WAITING_LATENCY, BACK_RECURSION, PREPARE_FWD_RECURSION, FWD_RECURSION_AND_STORE);
signal state, next_state: state_type;

begin
  llr_mem_instance: LLR_Memory port map(
    CLOCK => CLOCK,
    ENABLE => enable_llr_memory ,
    WRITE_ENABLE => llr_memory_write_enable,
    ADDRESS => llr_memory_address,
    DATA_IN => LLR,
    DATA_OUT => llr_memory_output
  );

-- this MUX is used because during the fwd recursion,
--the LLR used comes from the INPUT, and during the backward recursion,
--the LLR comes form the LLR memory.
```

```
recursion_llr_mux: process (LLR, llr_memory_output, recursion_llr_selector)
begin
  if (recursion_llr_selector = '1') then
    recursion_llr <= LLR;
  else
    recursion_llr <= llr_memory_output;
  end if;
end process;

H_memory: H_Matrix_ROM port map(
  CLOCK => CLOCK,
  ENABLE => enable_H_memory,
  ADDRESS_A => up_down_counter,
  DATA_OUT_A => H_column
);

RU: Recursion_Unit port map (
  CLOCK => CLOCK,
  RECURSION_H_COLUMN => H_column,
  LLR => recursion_llr,
  RESET_SM => reset_sm,
  WRITE_SM_REGISTERS => write_sm_registers,
  STATE_METRICS => state_metrics,
  PN_OUTPUT => pn_output
);

-- auxiliary counter used to access memory addresses
-- this counter counts from 0 to N-1 and from N-1 downto 0
-- values out of this range cause illegal address access
counter: process (CLOCK, reset_counter)
begin
   if (reset_counter = '1') then
      up_down_counter <= (others => '0');
  elsif falling_edge (CLOCK) then
    if (count_direction = '1') then
      if (up_down_counter /= conv_std_logic_vector(N-1, ADDRESS_BITS)) then
      -- this avoids illegal address access
        up_down_counter <= up_down_counter + '1';
      else
        up_down_counter <= up_down_counter;
      end if;
    else
      if (up_down_counter /= conv_std_logic_vector(0, ADDRESS_BITS)) then
      -- this avoids underflow
        up_down_counter <= up_down_counter - '1';
      else
        up_down_counter <= up_down_counter;
      end if;
    end if;
  end if;
end process;

-- transforms the signal state_metrics to be suitable
--for the Alpha memory input
alpha_values_mem_din_attribution: process(state_metrics)
begin
  for i in NUMBER_OF_SM -1 downto 0 loop
    alpha_values_memory_data_in((i+1)*SM_BITS -1 downto i*SM_BITS)
```

```
    <= state_metrics(NUMBER_OF_SM -1 -i);
  end loop;
end process;

process(alpha_values_memory_output)
begin
  for i in NUMBER_OF_SM -1 downto 0 loop
    alpha_values(NUMBER_OF_SM -1 -i)
    <= alpha_values_memory_output((i+1)*SM_BITS -1 downto i*SM_BITS);
  end loop;
end process;

Alpha_Memory: Alpha_Values_Memory port map(
  CLOCK => CLOCK,
  ENABLE => alpha_values_memory_enable,
  WRITE_ENABLE => alpha_values_memory_write_enable,
  ADDRESS => alpha_values_memory_address,
  DATA_IN => alpha_values_memory_data_in,
  DATA_OUT => alpha_values_memory_output
);

-- auxiliary counter
up_couter_process: process (CLOCK)
begin
  if rising_edge (CLOCK) then
    if (reset_up_counter =  '1') then
      up_counter <= (others => '0');
    elsif (count_up = '1') then
      up_counter <= up_counter + '1';
    else
      up_counter <= up_counter;
    end if;
  end if;
end process;

LLR_Unit_instance: LLR_Unit port map (
  CLOCK => CLOCK,
  ALPHA_VALUES => alpha_values,
  BETA_VALUES_1_TRANSITIONS => pn_output,
  BETA_VALUES_0_TRANSITIONS => state_metrics,
  LLR => llr_memory_output,
  WRITE_PIPELINE_REGS => write_pipeline_regs,
  RESET_MAP_REGISTER => reset_map_register,
  WRITE_RESULT => write_result,
  MAP_RESULT => map_result_signal
);

MAP_RESULT <= map_result_signal;

------------------------ FINITE STATE MACHINE -------------------------

process (CLOCK, RESET)
begin
  if (RESET = '1') then
    state <= RESET_STATE;
  elsif rising_edge(CLOCK)then
    state <= next_state;
  end if;
```

```
end process;

process (state, up_down_counter, up_counter)
begin
  case state is
    when RESET_STATE =>
      next_state <= FWD_RECURSION;
    when FWD_RECURSION =>
      if (up_down_counter  = conv_std_logic_vector(N-1, ADDRESS_BITS)) then
        next_state <= PREPARE_BACK_RECURSION;
      else
        next_state <= FWD_RECURSION;
      end if;
    when PREPARE_BACK_RECURSION =>
      next_state <= WAITING_LATENCY;
    when WAITING_LATENCY =>
      if (up_counter  = conv_std_logic_vector(1, ADDRESS_BITS)) then
        next_state <= BACK_RECURSION;
      else
        next_state <= WAITING_LATENCY;
      end if;
    when BACK_RECURSION =>
      if (up_down_counter  = conv_std_logic_vector(0, ADDRESS_BITS)) then
        next_state <= PREPARE_FWD_RECURSION;
      else
        next_state <= BACK_RECURSION;
      end if;
    when  PREPARE_FWD_RECURSION =>
      next_state <= FWD_RECURSION_AND_STORE;
    when FWD_RECURSION_AND_STORE =>
      if (up_down_counter  = conv_std_logic_vector(2, ADDRESS_BITS)) then
        next_state <= FWD_RECURSION;
      else
        next_state <= FWD_RECURSION_AND_STORE;
      end if;
    when others =>
      next_state <= RESET_STATE;
    end case;
end process;

-- control signals attribution
reset_sm <= '1' when (state = RESET_STATE or
state = PREPARE_FWD_RECURSION or state = PREPARE_BACK_RECURSION) else '0';
reset_map_register <= '0' when (state = BACK_RECURSION or
state = FWD_RECURSION_AND_STORE or state = PREPARE_FWD_RECURSION) else '1';
write_sm_registers <= '1' when (state = WAITING_LATENCY
or state = BACK_RECURSION or state = FWD_RECURSION_AND_STORE or
state = FWD_RECURSION) else '0';
--llr memory
recursion_llr_selector <= '1' when (state = FWD_RECURSION_AND_STORE or
state = FWD_RECURSION) else '0';
enable_llr_memory <= '0' when state = RESET_STATE else '1';
llr_memory_address <= up_counter when (state = FWD_RECURSION or
state = FWD_RECURSION_AND_STORE)  else up_down_counter;
llr_memory_write_enable <= '1' when (state = FWD_RECURSION or
state = FWD_RECURSION_AND_STORE or state = PREPARE_BACK_RECURSION) else '0';
-- Parity check matrix
enable_H_memory <= '1';
```

```
alpha_values_memory_enable <= '0' when (state = RESET_STATE) else '1';
alpha_values_memory_address <= llr_memory_address;
alpha_values_memory_write_enable <= llr_memory_write_enable;
write_pipeline_regs <= '0' when (state = RESET_STATE or
state = PREPARE_BACK_RECURSION or state = FWD_RECURSION) else '1';
--COUNTERS
reset_counter <= '1' when state = RESET_STATE or
state = PREPARE_FWD_RECURSION else '0';
count_direction <= '1' when (state = FWD_RECURSION or
state = FWD_RECURSION_AND_STORE or state = PREPARE_BACK_RECURSION) else '0';
reset_up_counter <= '1' when (state = RESET_STATE or
state = PREPARE_BACK_RECURSION or state = PREPARE_FWD_RECURSION) else '0';
count_up <= '1' when (state = WAITING_LATENCY or state = FWD_RECURSION or
state = FWD_RECURSION_AND_STORE or state = BACK_RECURSION)  else '0';
write_result <= '1' when (state = BACK_RECURSION or
state = FWD_RECURSION_AND_STORE or state = PREPARE_FWD_RECURSION) else '0';

end Behavioral;
```